

L'art de la visualisation graphique R avec ggplot2

La grammaire graphique en statistique, expliquée et appliquée

Version 1.7

Daname KOLANI

Contents

Préface

Introduction

R est devenu un langage un incontournable dans le domaine des sciences des données pour non seulement son potentiel en analyse de données mais également en visualisation des données. Pour visualiser les données avec R, nous disposons d'au moins trois systèmes de packages à savoir **graphics** pour réaliser les graphiques de base R, **lattice** qui permet de concevoir des graphiques conditionnels notamment et enfin **ggplot2** (où "gg" signifie "**G**rammar of **G**raphics") qui permet à travers une grammaire graphique, de concevoir des graphique couche par couche et qui offre également un haut niveau de contrôle sur les composants graphiques pour une visualisation de données très ergonomique et élégantes.
ggplot2 a hérité fondamentalement ses objets graphiques du package **grid**. Il est l'oeuvre de Hadley Wickham, qui n'est plus à présenter au vu du nombre impressionnant de packages à son actif à savoir **dplyr**, **tidyr**,**xml2**, **devtools**...

Le contenu du manuel

Dans ce manuel pratique, sur les fonctions et composants essentiels du package **ggplot2**, nous abordons d'abord dans le premier chapitre, la fonction **qplot()** pour une introduction à la visualisation graphique de bas niveau et ensuite dans le deuxième chapitre, nous traitons amplement des composants de la grammaire et du système d'objets graphiques **ggplot2** avec notamment comme fonction de base **ggplot()**. En fin le dernier chapitre est consacré essentiellement à une galerie graphique organisé en thème graphique. Où nous abordons en continue certains aspects ou paramétrages non usuel à travers des graphiques accompagnées de commentaires et de code source.

Précision importante : Malheureusement, **ggplot2** ne gère pas encore les graphiques en 3D

Pré-requis

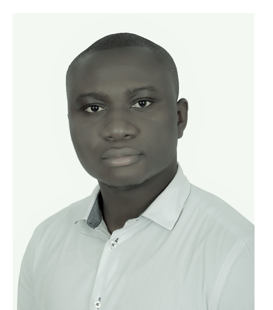
Pour utiliser ce manuel, il faut avoir maîtrisé les fondamentaux du langage R (syntaxe, fonctions et les structures de données ...) et une bonne connaisse des statistiques de base descriptive et exploratoire.
Matériellement parlant, une machine avec un système Microsoft Windows de préférence (n'importe quel OS supportant R suffira) avec le logiciel R installé ou mieux l'IDE **RStudio** en sus.

Les données et le code source

- Les données utilisées pour les différentes démonstrations dans les deux premiers chapitres sont essentiels des données built-in ou native du logiciel qui viennent avec les packages. Nous avons essentiel (pas seulement) utilisé le célèbre jeu de donnée diamonds livré avec **ggplot2**. Le dernier chapitre la galerie graphique utilise essentiellement le jeu de donnée de GapMinder. Mais tous les jeux données sont quand même fournis si l'on le désire dans un fichier nommé **Manuelggplot2.RData**.

- Le code source : Chaque graphique est toujours accompagné du code qui l'a généré, toutefois, si l'on dispose des fichiers sources, ils sont organisés en section. Ainsi, chaque fichier *.R source est nommé d'après son chapitre et la section spécifique.

À propos du rédacteur



KOLANI Daname, domicilié à ce jour à Casablanca (Maroc)

Fonction : Consultant Quantitatif

Site Web : www.ephiquant.com

Courriel : daname.kolani@ephiquant.com (Ephiquant SARL)

Actuellement, Consultant - Formateur aux outils quantitatifs à haut-potentiel **Ephiquant SARL**, je suis un passionné de la Data Science, de la finance, de l'économie, de l'économétrie, des statistiques & probabilités.
Mes langages favoris sont R et Python mais je pratique également d'autres langages comme VBA, MATLAB, C++, Java à titre de DSL...
Mes compétences concernent essentiellement les domaines suivants :
- La Data Science/Data Mining (data processing & wrangling , data Analysis, Visualisation, Modeling & Machine learning Algorithms ...)
- La finance de marché (gestion de portefeuille financier, l'évaluation d'actif financier, outils de gestion de risque...)
- La finance d'entreprise (Evaluation d'investissement & de financement, Analyse et diagnostic financier, Evaluation de société ...)
Je travaille et collabore avec **Vincent ISOZ** de Scientific-Evolution SARL (qui n'est plus à présenter) avec qui j'ai notamment écrit quelques ebooks. Il est en partie l'instigateur de la production ce manuel qui est en fait un support de formation.

Avertissements

Je décline toute responsabilité en cas de dommages corporels, matériels ou autres de quelque nature que ce soit, particuliers, indirects, accessoires ou compensatoires, résultant de la publication, de l'application ou de la confiance accordée au contenu du présent support. Je n'émets aucune garantie expresse ou implicite quant à l'exactitude ou à l'exhaustivité de toute information publiée dans le présent support, et ne garantit aucunement que les informations contenues dans cet ouvrage satisfassent un quelconque objectif ou besoin spécifique du lecteur. Je ne garantis pas non plus les performances de produits ou de services d'un fabricant ou d'un vendeur par la seule vertu du contenu du présent support.

En publiant des textes, il n'est pas dans l'intention principale du présent support de fournir des services de spécialistes ou autres au nom de toute personne physique ou morale ni pour mon compte, ni d'effectuer toute tâche devant être accomplie par toute personne physique ou morale au bénéfice d'un tiers. Toute personne utilisant le présent support devrait s'appuyer sur son propre jugement indépendant ou, lorsque cela s'avère approprié, faire appel aux conseils d'un spécialiste compétent afin de déterminer comment exercer une prudence raisonnable en toute circonstance. Les informations et les normes concernant le sujet couvert par le présent support peuvent être disponibles auprès d'autres sources que le lecteur pourra souhaiter consulter en quête de points de vue ou d'informations supplémentaires qui ne seraient pas couverts par le contenu du présent site Internet.

Je ne dispose (malheureusement...) d'aucun pouvoir dans le but de faire respecter la conformité au contenu du présent ouvrage, et je ne m'engage nullement à surveiller ni à faire respecter une telle conformité. Je n'exerce (à ce jour...) aucune activité de certification, de test ni d'inspection de produits, de conceptions ou d'installations à fins de santé ou de sécurité des personnes et des biens. Toute certification ou autre déclaration de conformité en matière d'informations ayant trait à la santé ou à la sécurité des personnes et des biens, mentionnée dans le présent support, ne peut aucunement être attribuée au contenu du présent support et demeure sous l'unique responsabilité de l'organisme de certification ou du déclarant concerné.

Votre avis nous intéresse!

En tant que lecteur de ce document, vous êtes le critique et le commentateur le plus important. Votre opinion compte et il est très intéressant de savoir ce qui est bien, ce qui peut être mieux et les sujets que vous souhaiteriez voir être traités. Vous pouvez m'envoyer un e-mail pour partager ce que vous avez aimé ou détesté dans le présent document afin d'en assurer une amélioration continue.
Email : daname.kolani@ephiquant.com

Remerciements

Je souhaiterais exprimer ma grande gratitude et remerciements à l'équipe qui maintient R et tous les développeurs des différents packages utilisés et cités dans le présent ouvrage notam-

ment Hadley Wickham , Paul Murrell, Baptiste Auguie, Kamil Slowikowski. . .

Liens Internet

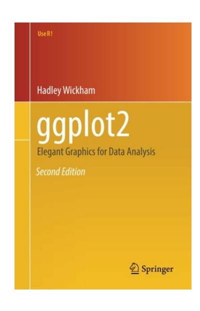
Documentation en ligne sur ggplot2 : http://docs.ggplot2.org/current/

ggplot2 Toolbox : https://rpubs.com/hadley/ggplot2-toolbox

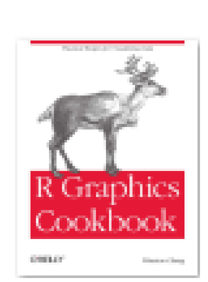
Themes with ggplot2 : http://www.hafro.is/~einarhj/education/ggplot2/themes.html

Scales, axes & legends with ggplot2 :
http://www.hafro.is/~einarhj/education/ggplot2/scales.html#sub-layers-legends

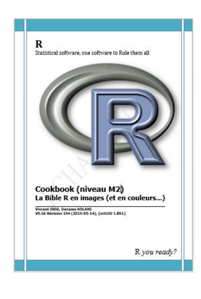
Médiagraphie



Titre : **Elegant Graphics for Data Analysis**
Auteur : **Hadley Wickham**
Caractéristiques : 260 Pages/Éditions Springer/
ISBN : 331924275X
Commentaire : Un livre écrit par l'auteur de **ggplot2** lui-même, qui est un essai sur les différentes fonctionnalités du package. Un livre assez descriptif, abordant de manière concise tous les aspects de la grammaire graphique.



Titre : **R Graphics Cookbook**
Auteur : **Winston Chang**
Caractéristiques : 413 pages / Éditions O'Reilly/
ISBN: 9781449316952
Commentaire : Un bon livre pour passer des graphiques de base R au package ggplot2. Le livre est écrit sous un style FAQ avec des questions, solutions et discussions.



Titre : **La Bible R en images (et en couleurs. . .)**
Auteurs : **Vincent ISOZ & Daname KOLANI**
Caractéristiques : 1 626 pages
Commentaire : Un véritable zoo de R. Ce livre aborde graduellement les concepts de la programmation R avant d'entamer des concepts avancés en Data Mining, Finance Quantitative . . .

1

Visualisation graphique de base avec qplot()

Contents

qplot() qui signifie littéralement **q**uick **plot** est la fonction de base qu'offre la librairie **qqplot2** pour le tracé de graphiques intuitifs et simples. Elle est pour **ggplot2** ce que la fonction **plot()** est pour **graphics** (la librairie graphique de base de R). Elle a été conçue pour non seulement permettre à l'utilisateur de réaliser rapidement des graphiques communiquant l'essentiels des caractéristiques d'un graphiques(les axes, les étiquettes, légendes...) mais également à titre de préliminaire pour l'appréhension de la grammaire graphique de **ggplot2**. Elle est convient parfaitement aux habitués de la fonction **plot()**.
Par ailleurs, elle a l'avantage de permettre également l'accès à l'ergonomie de **ggplot2** avec le minimum de complexité ou de maîtrise possible de la grammaire de cette dernière. Dans ce chapitre, nous étudierons d'abord l'essentiel des paramètres graphiques relatifs à cette fonction et ensuite nous allons découvrir comment réaliser les différents types de graphiques communément utiliser pour visualiser les données.

1.1 La structure minimale de la fonction qplot()

Avant d'aller plus loin, notons d'ores et déjà que même si la fonction **qplot()** est sensée restée aussi intuitive que la fonction **plot()**, structurellement, elles sont différentes. En effet, la fonction **qplot()** n'étant pas une fonction générique comme **plot()** de Classe S3, elle n'accepte comme structure de données que les **data.frame**.
Pour réaliser un graphique minimal comme celle de la figure 1.1. Nous devons obligatoirement fournir l'argument **x** et **y** ou mieux si ces derniers sont des vecteurs d'un data.frame on mettra celui-ci en argument pour **data**.

```
> # charger le package
> library(ggplot2)
> # charger le jeu de données "diamonds"
> data(diamonds)
> str(diamonds) # voir la structure interne des données
Classes 'tbl_df', 'tbl' and 'data.frame':
          53940 obs. of  10 variables:
 $ carat  : num  0.23 0.21 0.23 ...
 $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..:  ...
 $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..:  ...
 $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..:  ...
 $ depth  : num  61.5 59.4 ...
 $ table  : num  55 61 65 58 ...
 $ price  : int  326 326 327 334 335 ...
 $ x      : num  3.95 3.89 4.05 ...
 $ y      : num  3.98 3.84 4.07 4.23 ...
 $ z      : num  2.43 2.31 2.31 2.63 ...
> qplot(diamonds$carat,diamonds$price)
> # ou
> qplot(carat, price, data = diamonds)
```

Par défaut, le graphique produit un nuage de point. Si l'on veut obtenir un graphique univarié il va de soit qu'il faut omettre **y**.

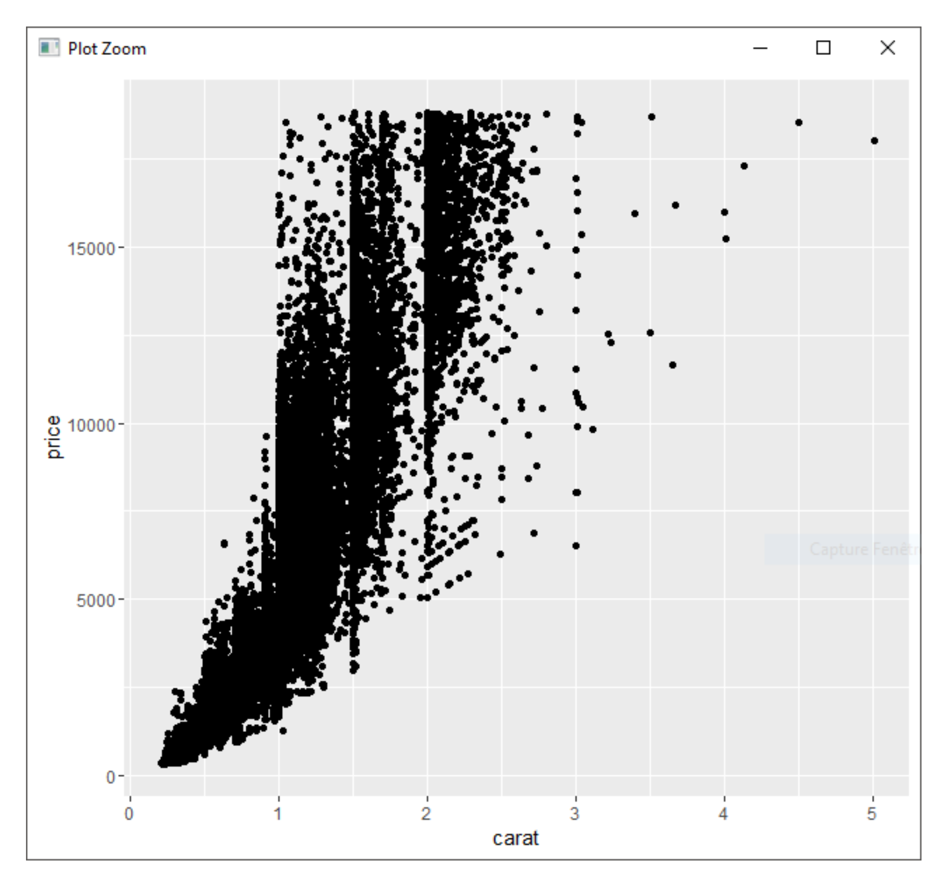


Figure 1.1 – Graphique minimal avec **qplot()**

Par ailleurs, la fonction **qplot()** autorise des transformations de variables telles que :

```
> # log transformation
> qplot(log(carat), log(price), data = diamonds)
> # opération de calcul en argument
> qplot(price, x*y*z, data = diamonds)
```

1.2 Les paramètres Aesthetiques : couleurs, forme, taille ...

Ce sont des paramètres qui nous permettent de modifier la couleur, la forme, la taille des objets graphiques à représenter. A ce niveau, lorsque l'une de ces paramètres est spécifiée, **qplot()**, génère automatiquement une légende pour mettre en évidence les spécificités Aesthetiques.

Les couleurs et formes

Pour agir sur la couleur des objets graphiques, on utilise l'argument **colour** et **shape** pour modifier la forme :

```
> # extraire un échantillon de 500 diamants
> db = diamonds[sample(nrow(diamonds),500),]
> # définir les couleurs selon l'échelle des niveaux de la
> # la variable catégorielle color
> qplot(carat, price, data = db, colour = color)
> # défin r les formes à l'échelle du facteur cut
> qplot(carat, price, data = db, shape = cut)
> # définir coulour et forme à l'échelle de variable
> qplot(carat, price, data = db, colour = color, shape = cut)
```

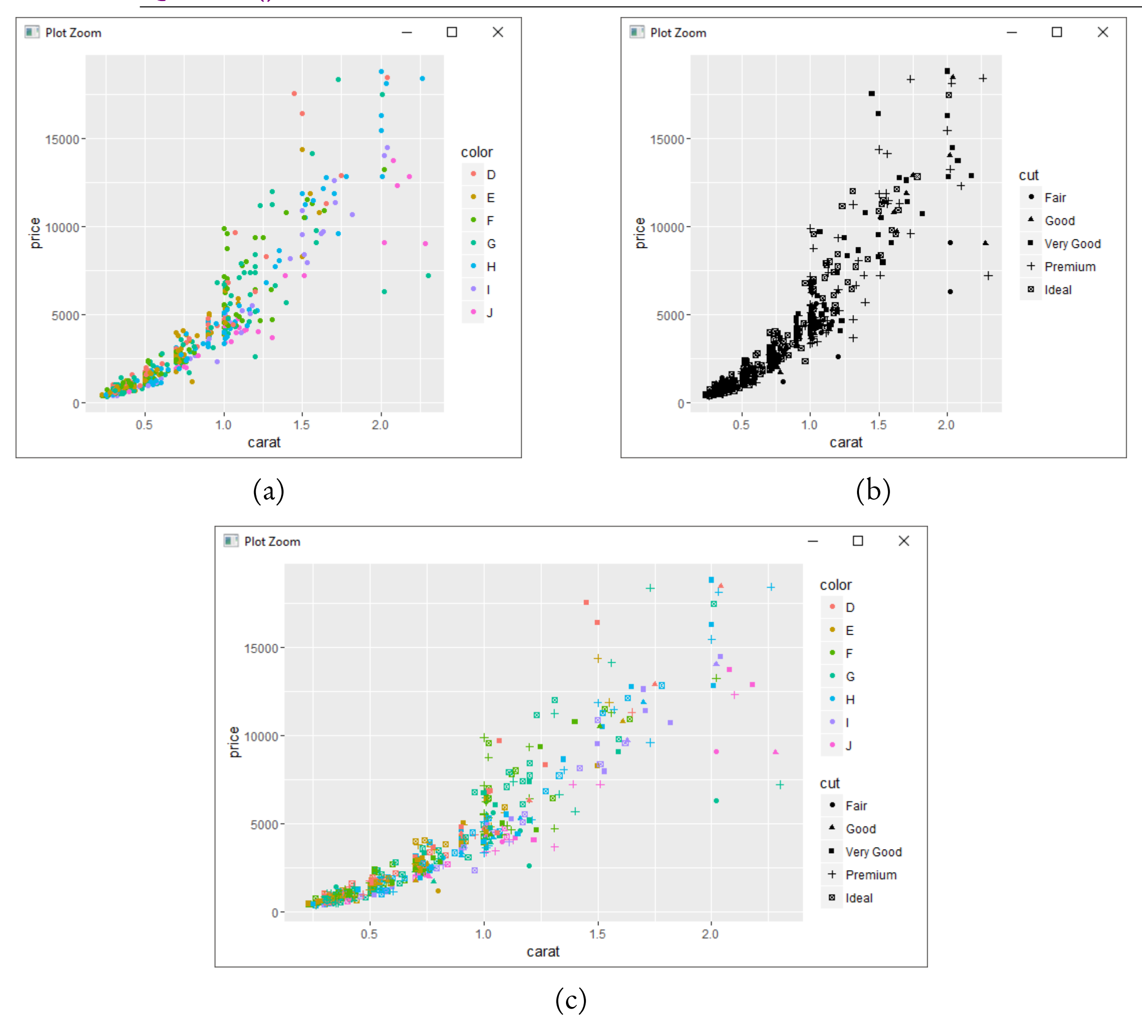


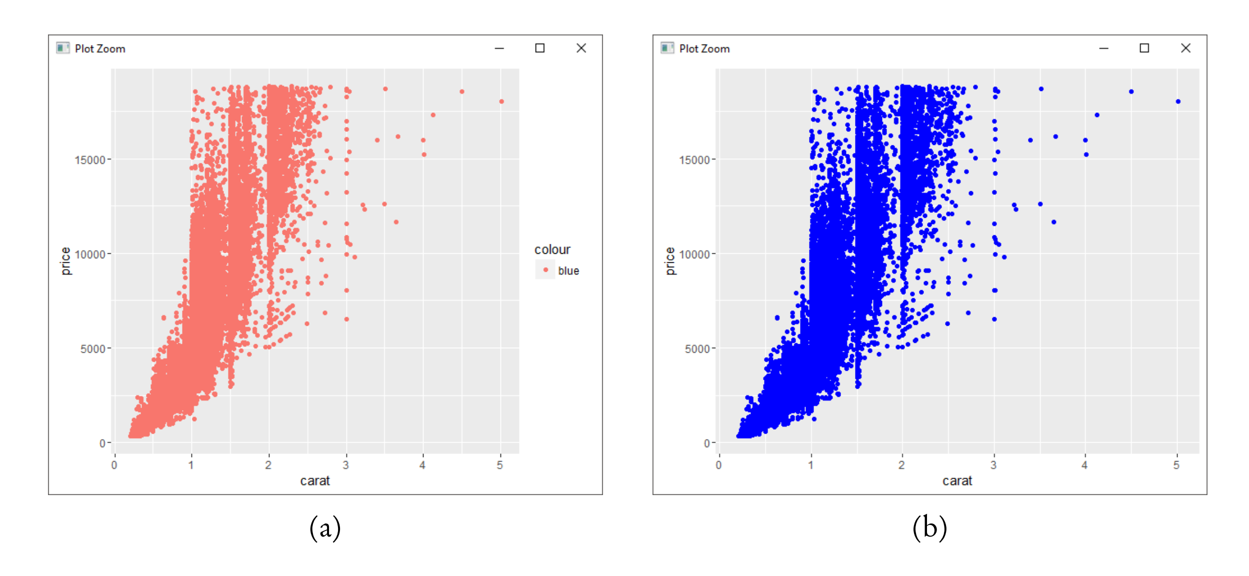
Figure 1.2 – Couleurs et formes

Remarquez que les variables fournies en argument pour la couleur et la forme sont de type catégoriel, des facteurs. Et pour la plupart des paramètres Aesthetiques, **ggplot2** utilise en arrière plan la fonction **scale** pour générer une palette de couleur ou de forme ou le taille...

Spécification manuelle d'Aesthetiques avec I()

Imaginons que nous voulons spécifier une couleur ou une forme ou une taille, nous la passons en argument de la fonction **I()** avant de passer cette dernière en paramètre. Ainsi, pour spécifier par exemple la couleur *blue* au nuage point on fera comme ceci :

```
> # cette écriture n'a aucun effet
> qplot(carat,price,data = diamonds,colour = "blue")
> # par contre avec I()
> qplot(carat,price,data = diamonds,colour = I("blue"))
```



(a) (b)

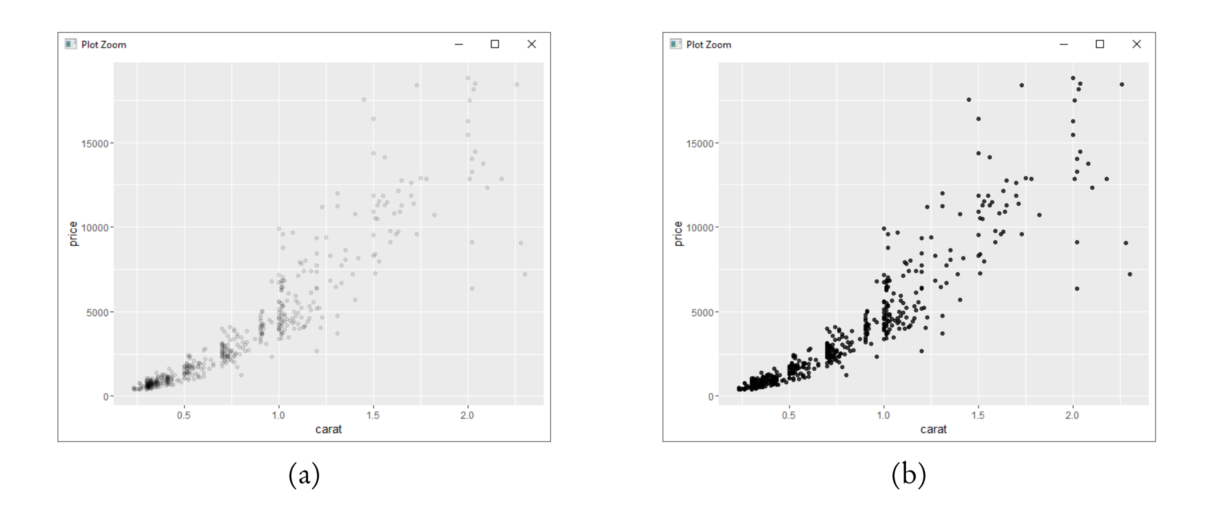
Figure 1.3 – Spécification manuelle d'aesthetique

La fonction **I()** ne s'applique pas qu'aux couleurs mais également aux autres paramètres Aesthetiques comme nous allons le voir par la suite.

Modifier la Transparence

Pour agir sur l'opacité ou la transparence des points ou des objets graphiques, on utilise l'argument **alpha** qui prend une valeurs comprise entre 0 (transparence) et 1 (opacité totale)

```
> qplot(carat,price,data = db,alpha = I(1/10))
> qplot(carat,price,data = db,alpha = I(3/4))
```



(a) (b)

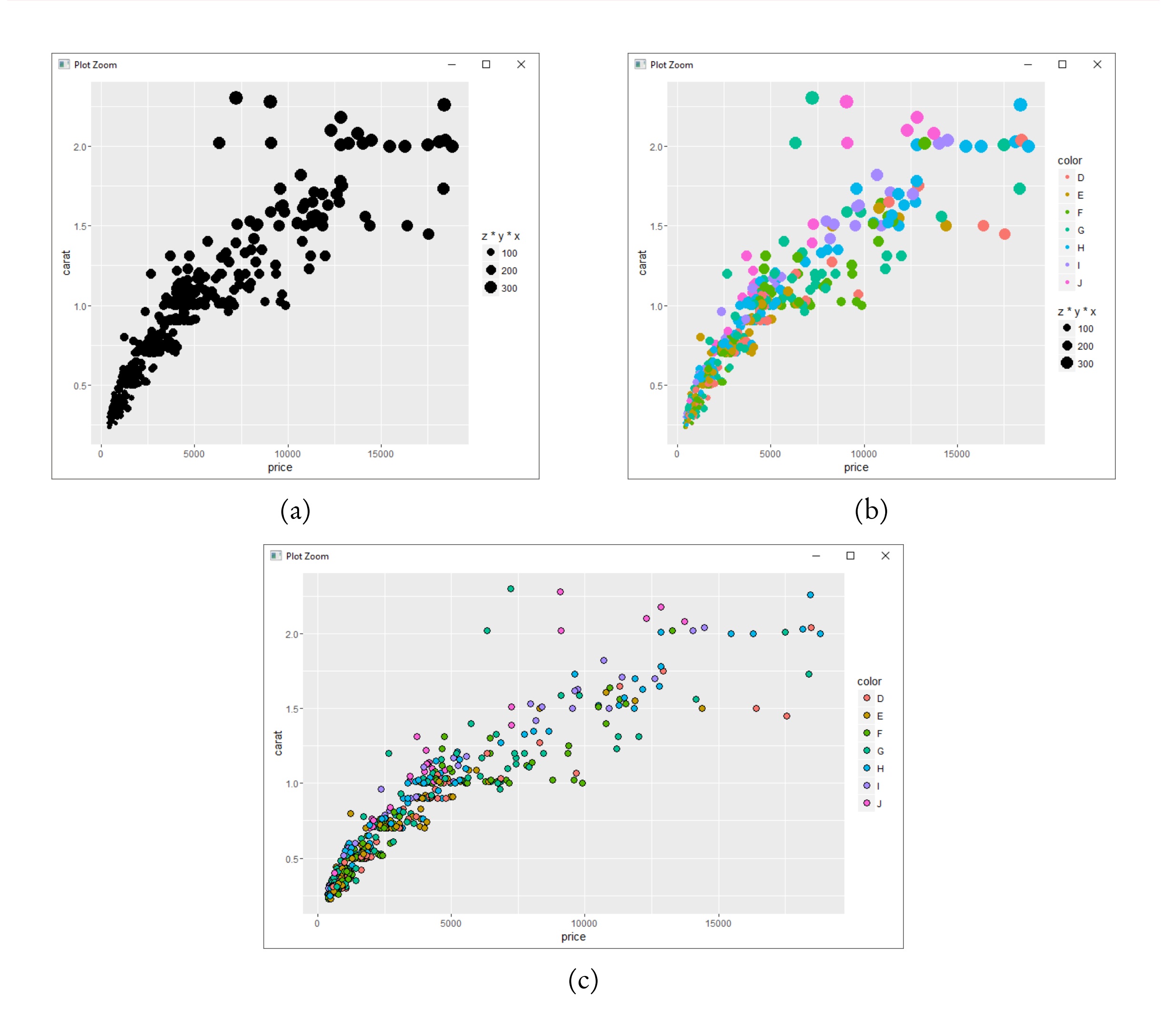
Figure 1.4 – Modification de la transparence ou opacité

Autres paramètres Aesthetics

Autres paramètres utiles, nous avons **fill** pour définir la couleur de remplissage de certains objets graphiques et **size** pour spécifier la taille.

```
> # spécification de taille
```

```
> qplot(price, carat, data = db, size = z*y*x)
> # Taille et coulour
> qplot(price, carat, data = db,colour = color, size = z*y*x)
> # Ajout de remplissage, specification manuelle de forme...
> qplot(price,carat , data = db,colour = I("black"),
      size = I(3), shape=I(21), fill=color)
```



(a) (b) (c)

Figure 1.5 – Autres paramètres de remplissage et taille

Tout comme avec la couleur ou la forme, on peut définir la couleur de remplissage **fill** ou la taille **size** avec une variable du jeu de données ou avec une constante (donc une spécification manuelle) à travers la fonction **I()**, comme on peut le voir dans le code ci-dessus.

1.3 Titres, étiquettes et graduation des axes

Les paramètres utilisés ici sont très similaires à ceux de la fonction **plot()**.

L'exemple ci-dessous présente une utilisation de ces paramètres comme on peut le voir à la figure 1.6 :

main	Permet de définir le titre
xlab, ylab	ces arguments permettent de spécifier les étiquettes des axes
xlim, ylim	ils permettent de spécifier un vecteur contenant la valeur minimale et maximale de l'axe des abscisses(xlim) et/ou des ordonnées(ylim)
log	cet argument permet de mettre les axes à l'échelle logarithmique, log="x" ou log="xy"

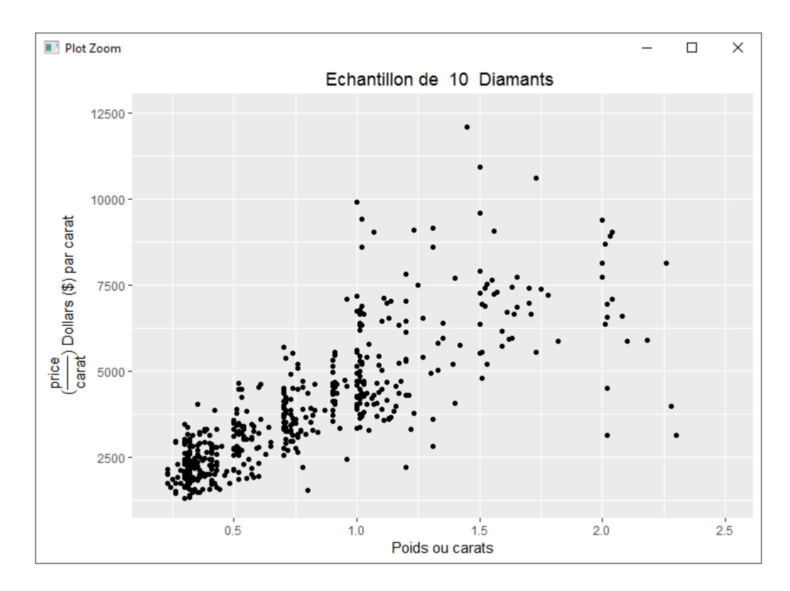


Figure 1.6 – Titres, étiquettes ...

```
> qplot(carat, price/carat, data = db,
 ylab = expression(paste((frac(price,carat)),
        " Dollars ($) par carat")),
 xlab = "Poids ou carats",
 main=paste("Echantillon de ",length(db)," Diamants "),
 xlim = c(0.2,2.5), # définir les limites de l'axe des x
 ylim = c(1300,12500)) # définir les limites de l'axe des y
```

La fonction **expression()**, sert à écrire les formules ou expressions mathématiques, pour en savoir d'avantage **?expression**.

1.4 Facetting ou vignette

Lorsque l'on dispose de données multivariées, on a d'abord l'option des Aesthetiques pour mettre en évidence les différents sous-ensembles ou groupes. Mais nous avons une autre option celle des facets ou vignettes qui nous permet d'afficher dans une même fenêtre graphique plusieurs graphiques correspondants aux différents sous-ensembles:

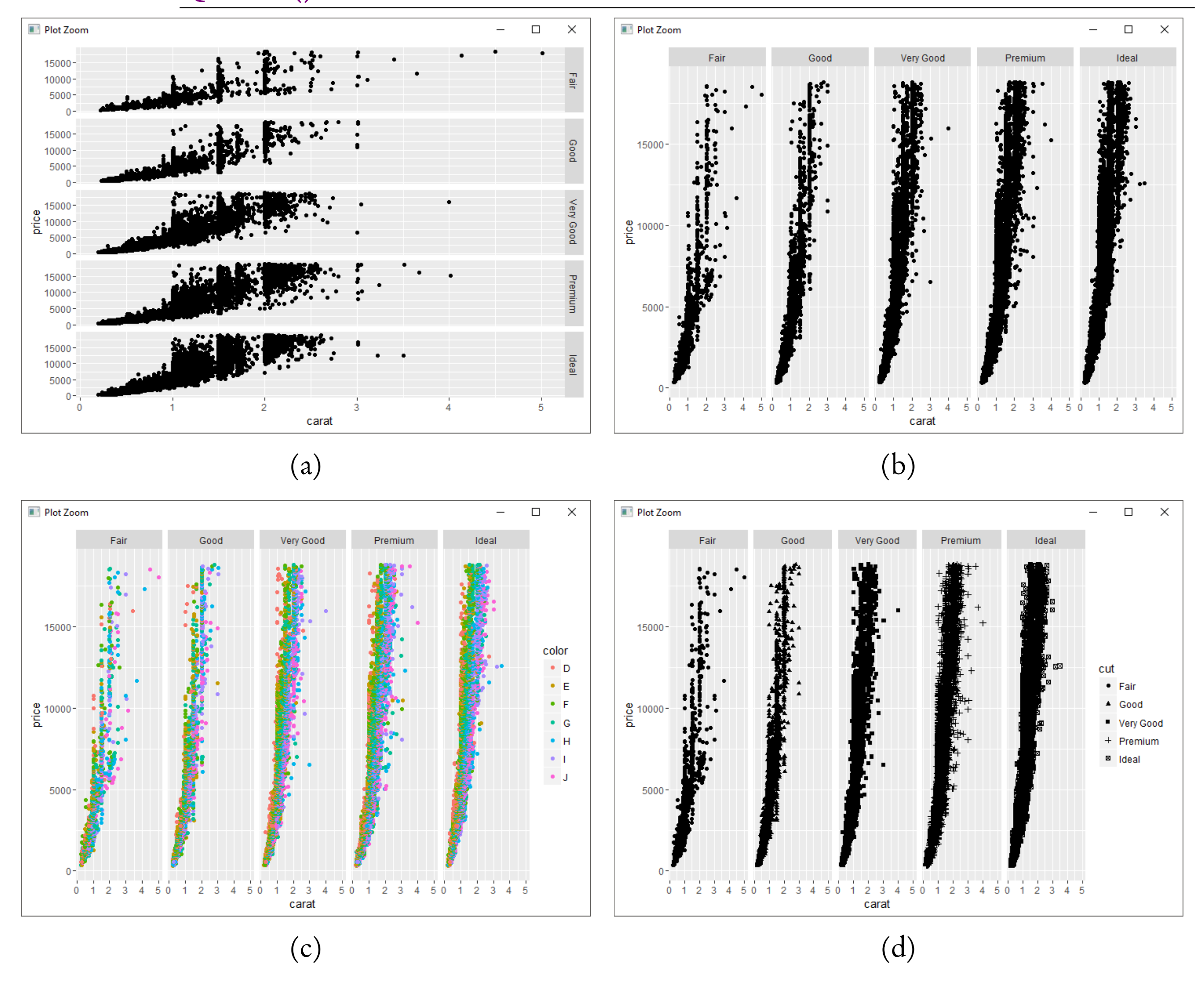


Figure 1.7 – Combinaisons facets et Aesthetiques

```
> # subdivision horizontale selon la coupure(cut) du diamant
> qplot(carat, price, data = diamonds,facets = cut~.)
> # subdivision verticale
> qplot(carat, price, data = diamonds,facets = .~cut)
> # combinaison facetting & Aesthetiques
> qplot(carat, price, data = diamonds,facets = .~cut,
          colour = color)
> qplot(carat,price, data = diamonds,facets = .~cut,
           shape= cut)
```

Dans les exemples suivants 1.7, nous avons utiliser la variable **cut** avec l'argument **facets** pour générer des sous-ensembles. Par ailleurs, nous avons également varié des paramètres Aesthetiques en vue de montrer la différence entre ces deux types de paramètres. **facets** impacte la disposition des figures graphiques alors que les paramètres Aesthetiques impactent les caractéristiques(couleur, forme, taille…) des objets graphiques.

1.5 Le paramètre *geom*

qplot(), n'est pas limité à produire seulement des nuages de points. En effet, son argument **geom** abréviation de géometric, permet de créer les objets graphiques de notre choix notam-

ment histogramme, courbes, diagramme à moustaches, diagramme en barres .... Le type d'objet géométrique choisi doit être en accord avec les données fournies (uni ou bi-variées).

Diagramme en barres

Pour obtenir un diagramme en barres, il faut spécifier **geom**="bar". Outres le paramètres Aesthetiques nous avons **weight**, qui permet au lieu d'utiliser un comptage habituel des classes comme le montre la figure 1.8a, utilise le comptage de classe pondéré par le nombre de carat par couleur comme le montre la figure 1.8b.

```
> qplot(color, data = db, geom = "bar")
> # modifier les couleurs de remplissages et de contour
> qplot(color, data = db, geom = "bar",
      # weight permet de pondérer chaque classe par une
         variable
      # ici par le nombre de carat
      weight = carat, ylab = "carat",
      fill = I("white"),colour = I("black"))
> qplot(cut, data = db, geom = "bar",fill=color,
     colour =I("black"))
> # ajout de facets
> qplot(color, data = db, geom = "bar",
      weight = carat, ylab = "carat",
      fill = color,colour = I("black"),facets = .~cut)
```

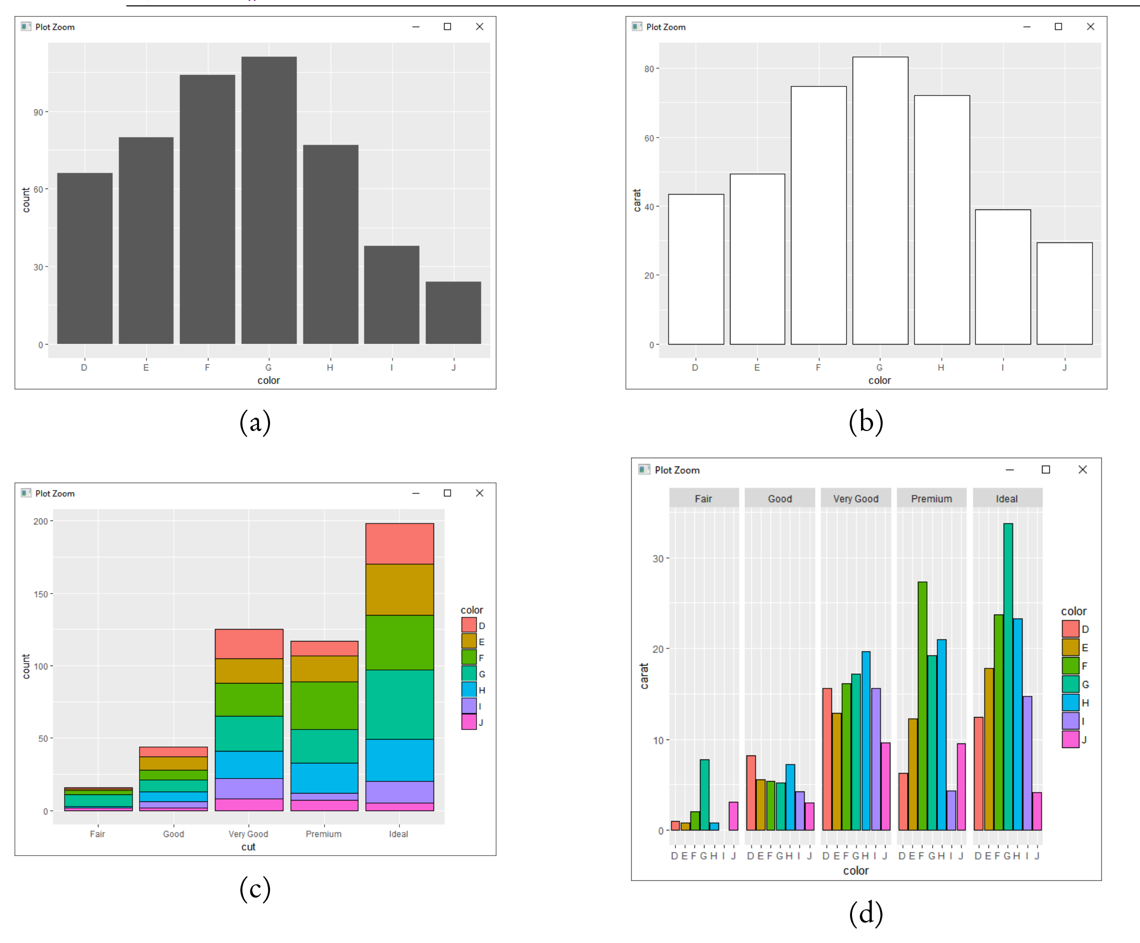


Figure 1.8 – Barchart ou diagramme en barres

Nous verrons comment contrôler l'ordre des barres des diagrammes en barres lors de l'étude de la fonction ggplot().

Histogrammes et courbes de densité

La construction d'un histogramme (1.9) nécessite que **geom** prenne comme valeur "histogram". De plus, outres les paramètres Aesthetiques, nous avons des paramètres propres aux histogrammes tels que les intervalles avec **breaks** ou la largeur des intervalles **binwidth**.

```
> qplot(depth,data = diamonds,geom="histogram",
      colour = I("white"), xlab="$ par cm cube")
> # spécifier des intervalles manuellement
> itvl <- with(diamonds,seq(min(depth),max(depth),
            length.out = 50))
> qplot(depth,data = diamonds,geom="histogram",
      xlab = "$ par cm cube",colour = I("red3"),breaks=itvl)
> # spécifier la largeur des barres
> qplot(depth,data = diamonds,geom="histogram",
      xlab ="$ par cm cube",colour = I("red3"),
      fill = I("steelblue"),binwidth=0.3)
> qplot(depth,data = diamonds,geom="histogram",
      xlab ="$ par cm cube", colour = I("gray"),
```

```
        fill = color,binwidth=0.25)
```

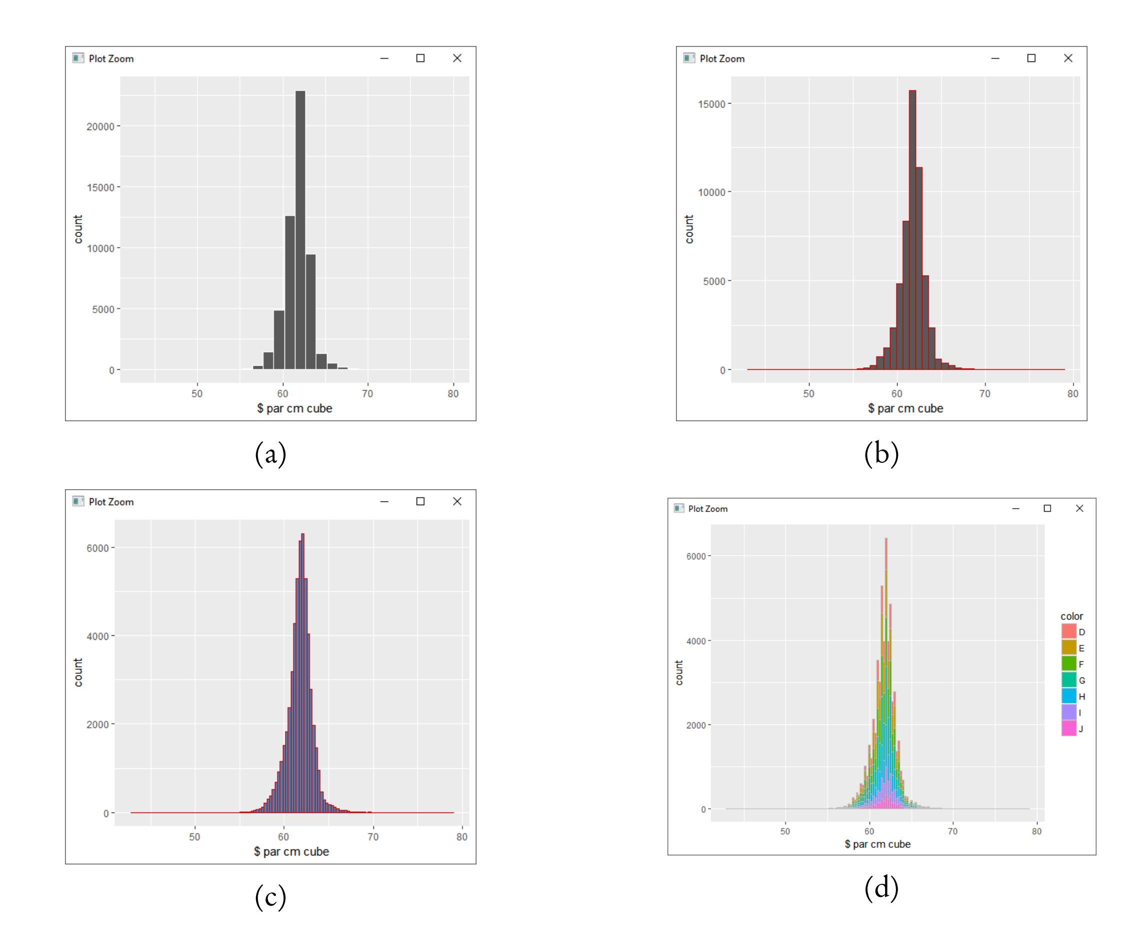


Figure 1.9 – Histogrammes avec **qplot()**

Le paramètre **breaks** peut prendre une valeur pour spécifier le nombre d'intervalles ou un vecteur pour spécifier le découpage de la distribution.
Pour la courbe de densité (1.10) nous pouvons utiliser **geom**="density" pour une vraie densité ou **geom**="freqpoly" pour générer un polygone de fréquence (1.10b).

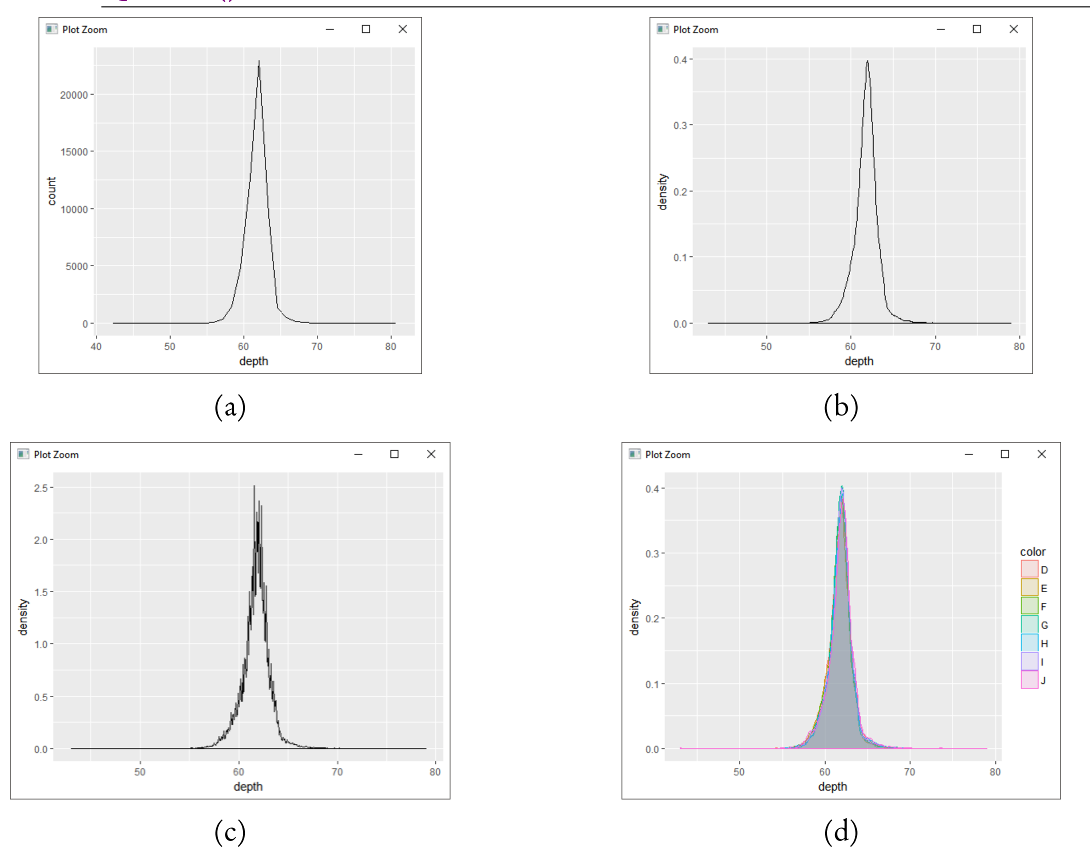


Figure 1.10 – Polygone de fréquence et courbes de densité avec **qplot()**

```
> # polygone de fréquence
> qplot(depth,data = diamonds, geom="freqpoly")
> qplot(depth,data = diamonds, geom = "density",binwidth =
  0.05)
> # ajuster le lissage de la courbe
> qplot(depth,data = diamonds, geom = "density",
      adjust = 0.05,binwidth = 0.05)
> qplot(depth,data=diamonds, geom = "density",
      colour= color,alpha=I(0.15),binwidth = 0.05,fill=color)
```

Le paramètre **adjust** permet de d'améliorer le niveau de lissage ou régularité (smoothness) de la courbe. Le paramètre **binwidth** ainsi que d'autres paramètres Aesthetiques sont applicables.
Pour la combinaison des deux types de graphiques (histogramme & courbe de densité), voir une illustration à la figure 2.22a.

Diagrammes Box Plots et jitter ou gigue

Ces deux diagrammes permettent de visualiser la distribution de variables quantitatives continues en fonction d'un caractère catégoriel.

```
> qplot(cut,carat, data = diamonds, geom =  "boxplot")
> qplot(cut,carat, data = diamonds, geom =  "jitter",
      alpha = I(0.2)) # définir lz dégré de transparence
```

```
> qplot(cut,carat,data = diamonds[diamonds$color==c("E","I")
  ,],
      geom = "boxplot",fill = color)
> qplot(cut,carat,data = diamonds,geom=c("jitter","boxplot"))
```

(a)

(b)

(c)

(d)

Figure 1.11 – Diagramme Boxplot et jitter ou gigue

Comme on peut le voir sur la figure 1.11, on peut faire **geom** = "boxplot" pour obtenir un diagramme Boxplot (1.11a) ou **geom** = "jitter" pour obtenir une gigue (1.11b) ou les deux en même temps (1.11d) avec **geom** = c("jitter", "boxplot"). On peut vouloir révéler un troisième caractère catégoriel en utilisant le paramètre aesthetique **fill**.

Diagrammes à points catégoriels ou Cleveland

Nous pouvons obtenir également des diagrammes à points catégoriels. Pour que cela fonctionne correctement, il faut nommer les lignes des données avec une variable catégorielle.

```
> data("precip")
> str(precip)
 Named num [1:70] 67 54.7 7 48.5 14 17.2 20.7  ...
 - attr(*, "names")= chr [1:70] "Mobile" "Juneau" ...
> val = precip[-50]; val = val[order(val)][1:20]
```

```
> lab = names(val); lab = factor(lab, levels = lab)
> qplot(val, lab, data = data.frame(lab,val))
```

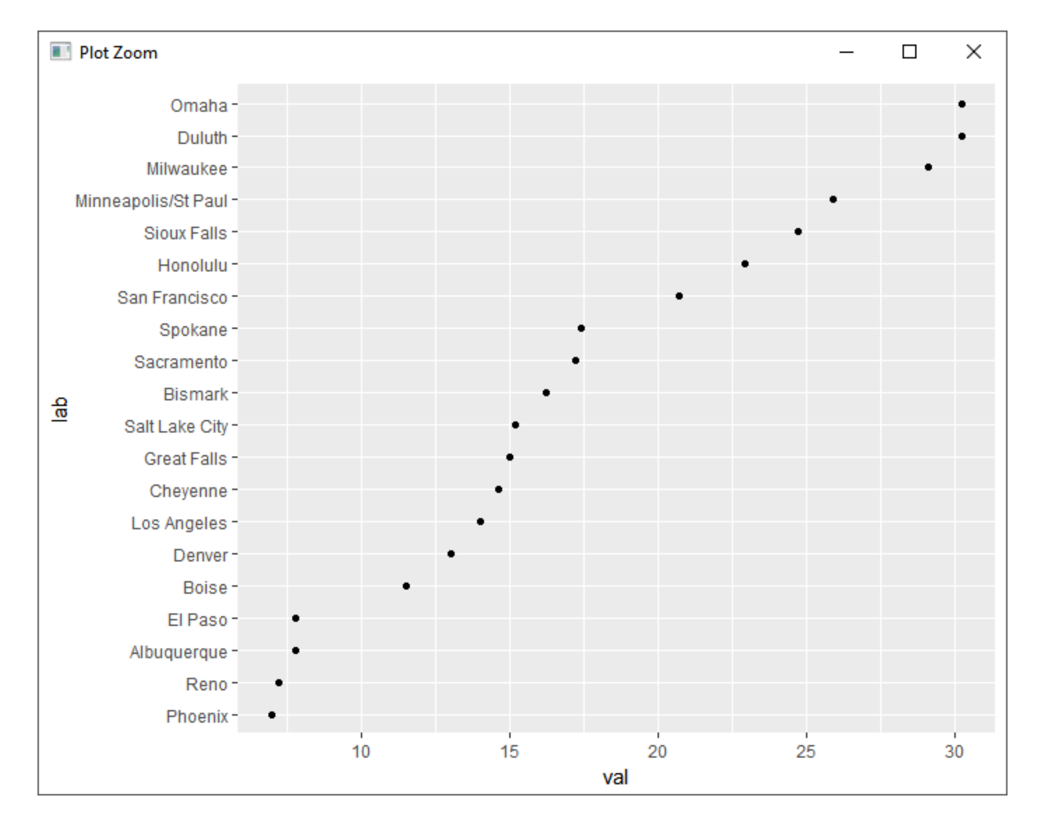


Figure 1.12 – Diagramme à points

Tracés de courbes ou lignes et Visualisation de Time Series

Pour obtenir un tracé de courbe, on définit **geom**="line" (figure 1.13b) ou "path" (figure 1.13c). La différence entre ces deux types de courbe, c'est que la première joint les données dans l'ordre en allant de la gauche vers la droite, alors que la deuxième joint les données dans l'ordre dans laquelle elles sont stockées.

```
> data("economics")
> str(economics)
Classes 'tbl_df', 'tbl' and 'data.frame': 574 obs. of 6
   variables:
 $ date    : Date, format: "1967-07-01" "1967-08-01" ...
 $ pce     : num  507 510 516 513 518 ...
 $ pop     : int  198712 198911 199113 199311 ...
 $ psavert : num  12.5 12.5 11.7 12.5 12.5 12.1 ...
 $ uempmed : num  4.5 4.7 4.6 4.9 4.7 4.8 5.1 ...
 $ unemploy: int  2944 2945 2958 3143 3066 3018 2878 ...
> year <- function(x) as.POSIXlt(x)$year + 1900
> economics$year <- year(economics$date)
> qplot(date, unemploy/pop, data = economics, geom = "line")
> qplot(date, unemploy, data = economics,
+ geom = c("line","point"))
> qplot(unemploy/pop, uempmed, data = economics,
```

```
+        geom = "path")
> qplot(unemploy/pop, uempmed, data = economics,
+        geom = c("point", "path"), color=year)
```

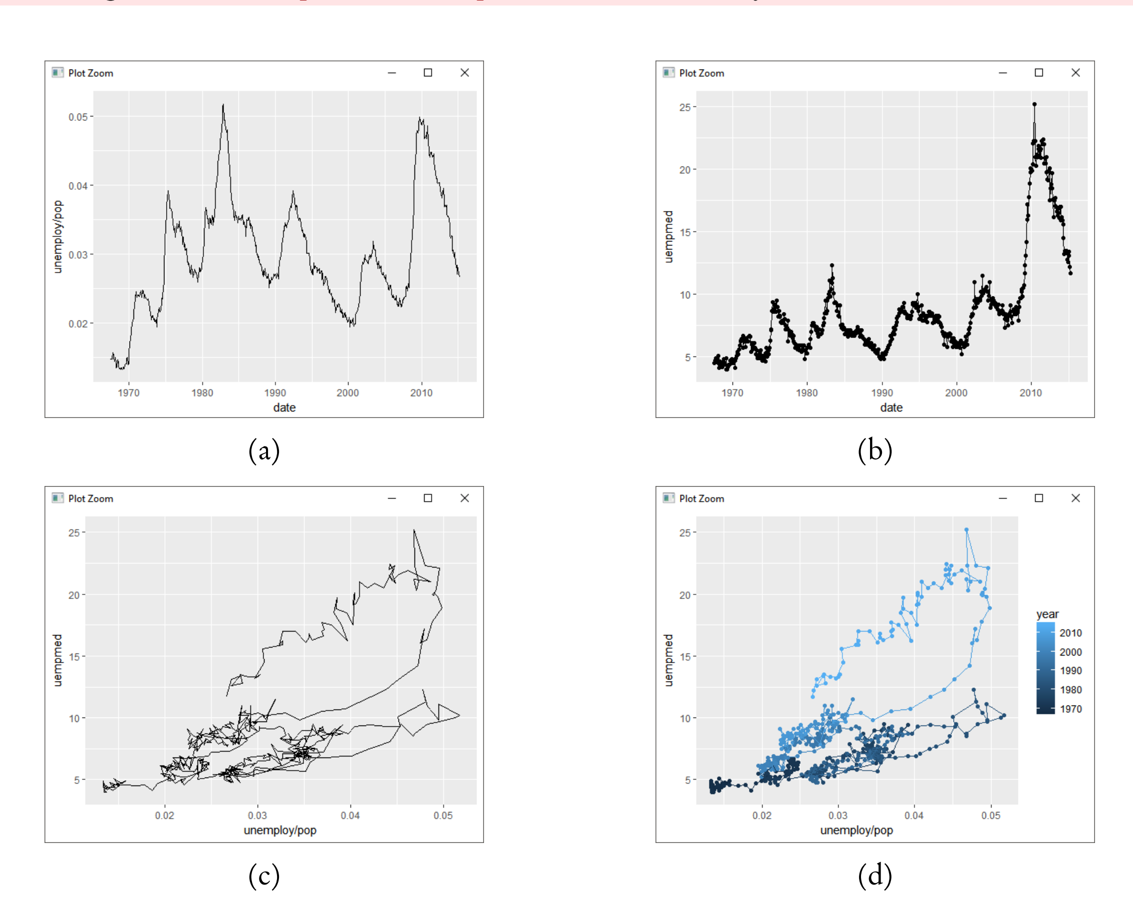


Figure 1.13 – Tracé de courbe

On peut combiner les courbes avec les nuages de points comme on peut le voir sur les figures 1.13a et 1.13d.

Les nuages de points

Par défaut, nous l'avons vu dans la section précédente, **qplot()**, génère automatiquement un nuage point lorsque **x** et **y**, sont fournis en paramètres. Toutefois, nous pouvons expliciter comme ceci **geom**="point". La figure 1.14 présentent les différentes valeurs que peut prendre le paramètre **shape**.

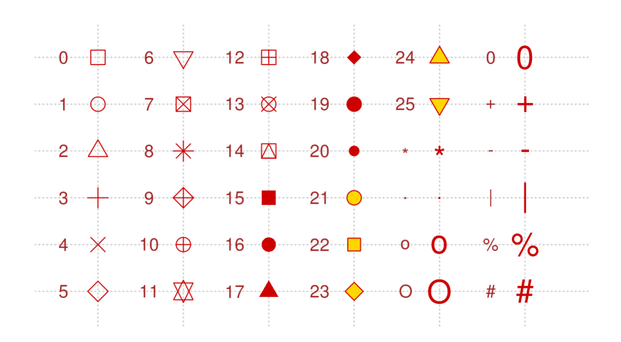
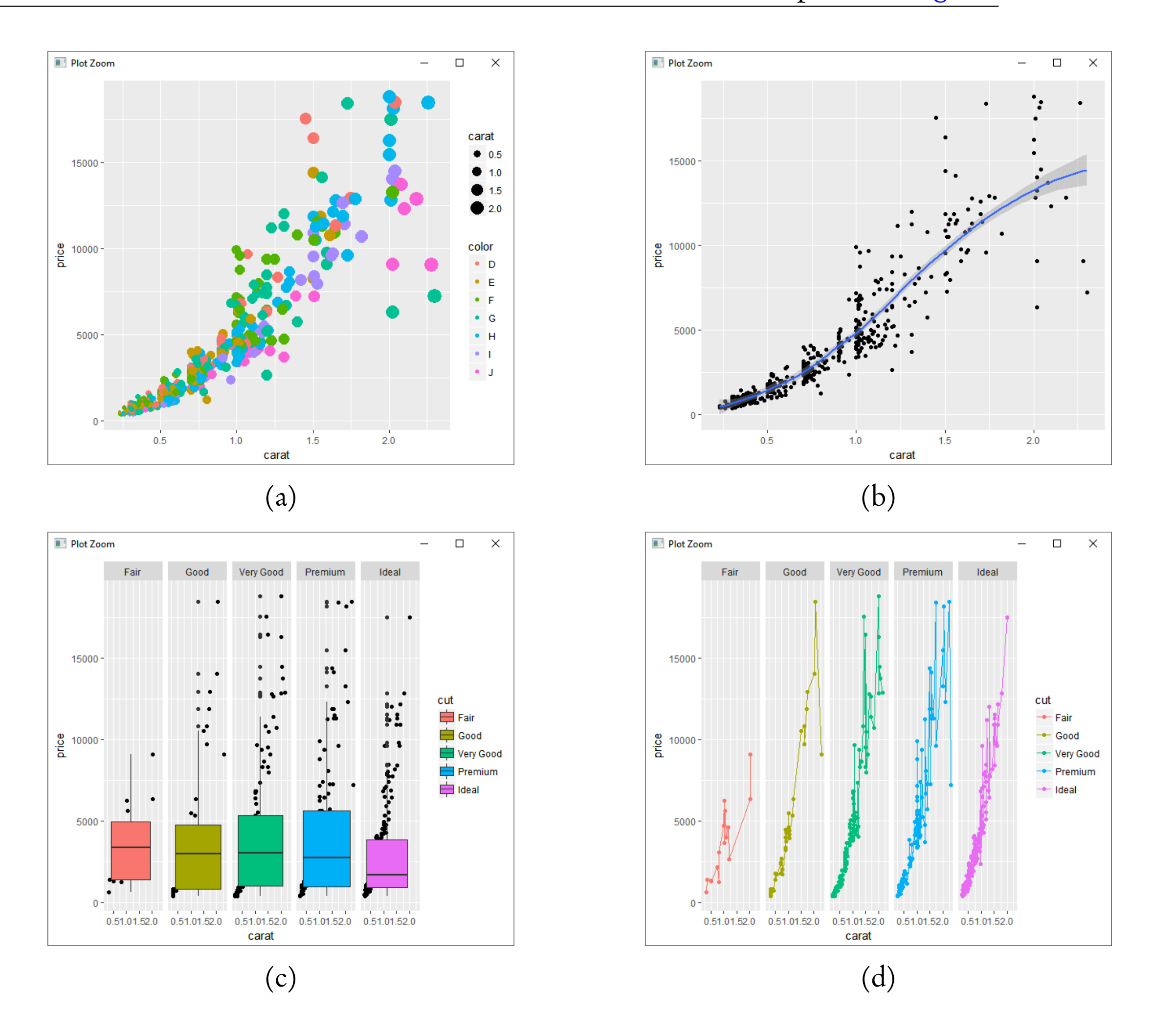


Figure 1.14 – Les différentes formes de points

Nous pouvons combiner plusieurs types d'autres formes géométriques ou d'objets graphiques avec les nuages de points comme on peut le voir à la figure 1.15.

```
> qplot(carat, price, data = db, colour = color, size= carat,
     shape = I(19))
> qplot(carat, price, data = db, geom=c("point", "smooth"))
> qplot(carat, price, data = db, geom=c("point", "boxplot"),
      facets =  .~cut,fill = cut)
> qplot(carat, price, data = db, geom=c("point", "line"),
     facets =  .~cut,colour = cut)
```

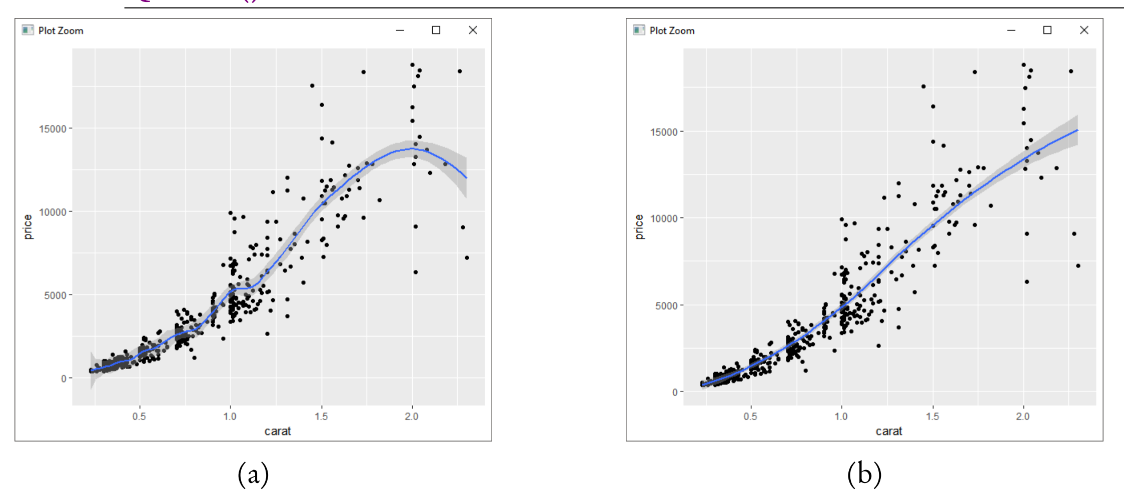


(a) (b) (c) (d)

Figure 1.15 – Les nuages de points avec d'autres objets graphiques

Un autre aspect particulier qu'on peut relever, c'est la combinaison d'une courbe d'ajustement avec une bande de confiance autour à la figure 1.15b. Par défaut, c'est le modèle de régression non paramétrique **loess** qui est utilisée. Nous pouvons modifier le niveau de lissage de la courbe à travers le paramètre **span** qui prend une valeur compris en 0 et 1 comme ci-dessous (résultat à la figure 1.16) :

```
> # span argument varié
> qplot(carat, price, data = db, geom = c("point","smooth"),
      span = 0.2)
> qplot(carat, price, data = db, geom = c("point","smooth"),
      span = 1)
```



(a) (b)

Figure 1.16 – Variation du niveau de le lissage de la courbe d'ajustement avec **span**

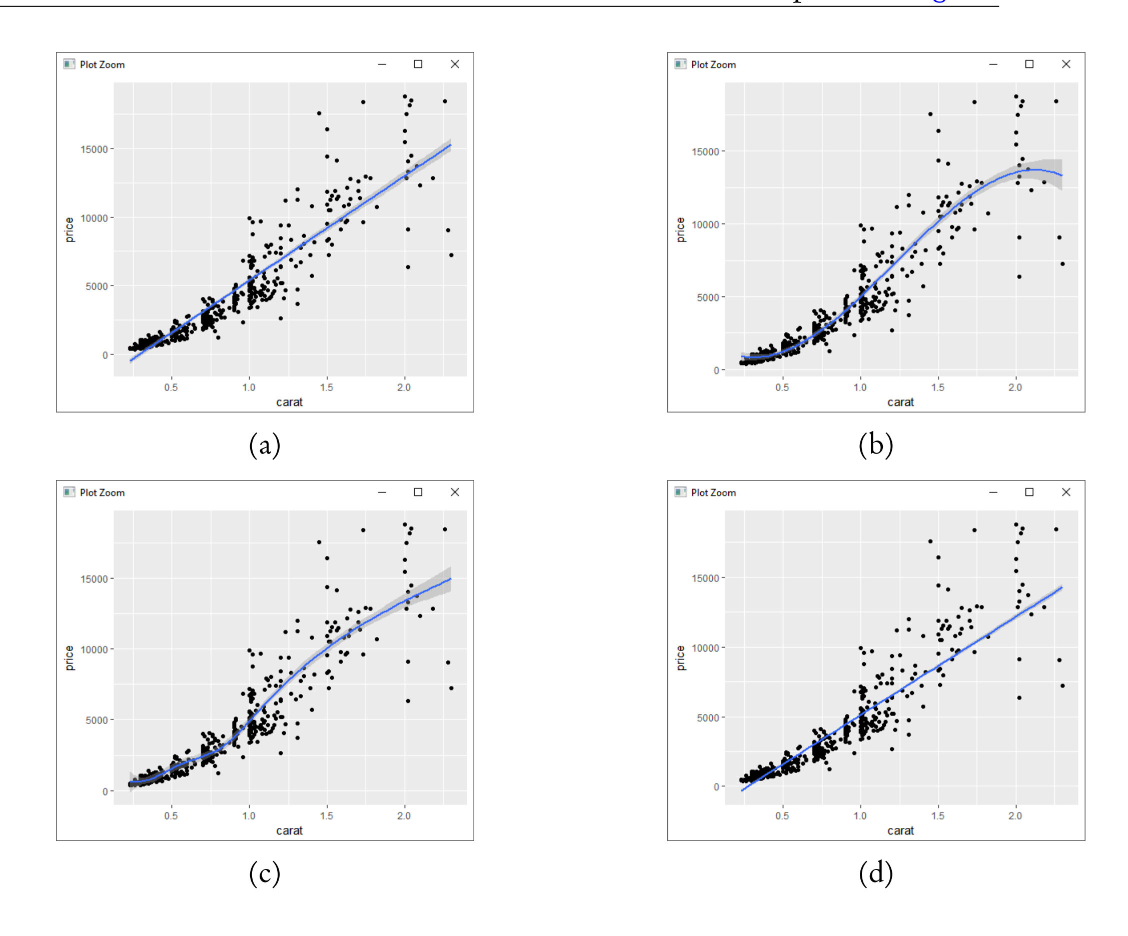
Par ailleurs, nous pouvons changer le modèle qui génère la courbe d'ajustement en utilisant le paramètre **method** et définir sa formule avec le paramètre **formula** :

```
> # modèle linéaire simple
> qplot(carat, price, data = db, geom=c("point", "smooth"),
     method="lm")
> # régression polynomiale degré 3
> qplot(carat, price, data = db, geom=c("point", "smooth"),
     method="lm", formula=y ~ poly(x,3))
> # modèle spline
> library(splines)
> qplot(carat, price, data = db, geom=c("point", "smooth"),
      method="lm", formula=y ~ ns(x,5))
> # modèle linéaire robuste avec gestion des valeurs extrêmes
> library(MASS)
> qplot(carat, price, data = db, geom=c("point", "smooth"),
        method="rlm")
```

Dans la figure ci-dessous 1.17, nous mettons en évidence quelques courbes d'ajustement notamment du modèle linéaire.

```
> library(mgcv)
> qplot(carat, price, data = db, geom = c("point", "smooth"),
                    method = "gam", formula = y ~ s(x))
```

Nous pouvons utiliser également le modèle additif généralisé **method**="gam" comme dans le code ci-dessus.



(a) (b)

(c) (d)

Figure 1.17 – Utilisation de différents modèles pour la courbe d'ajustement

Nous pouvons varier notre façon d'utiliser les facets et c'est plus significatif avec les nuages de points comme on peut le voir à la figure 1.18

```
> qplot(price, carat, data=diamonds, size = I(1.5),
      facets = cut ~ color)
```

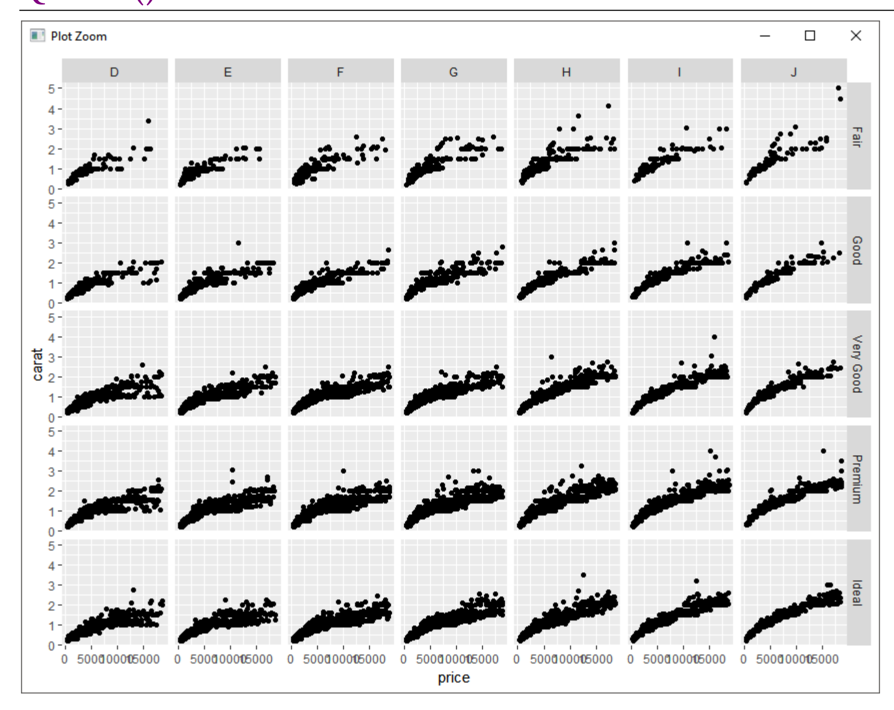


Figure 1.18 – Présenter 4 variables avec un nuage de point en utilisant **facets**

Pour obtenir le même graphique avec différentes couleurs pour chaque vignette ou facette, il faut fournir autant de couleurs qu'il y a de vignettes ou facettes. Ici, nous allons utiliser la fonction **interaction()** pour le mapping de couleur:

```
> qplot(price, carat, data=diamonds, size = I(1.5),
     facets = cut ~ color,color = interaction(cut,color))
```

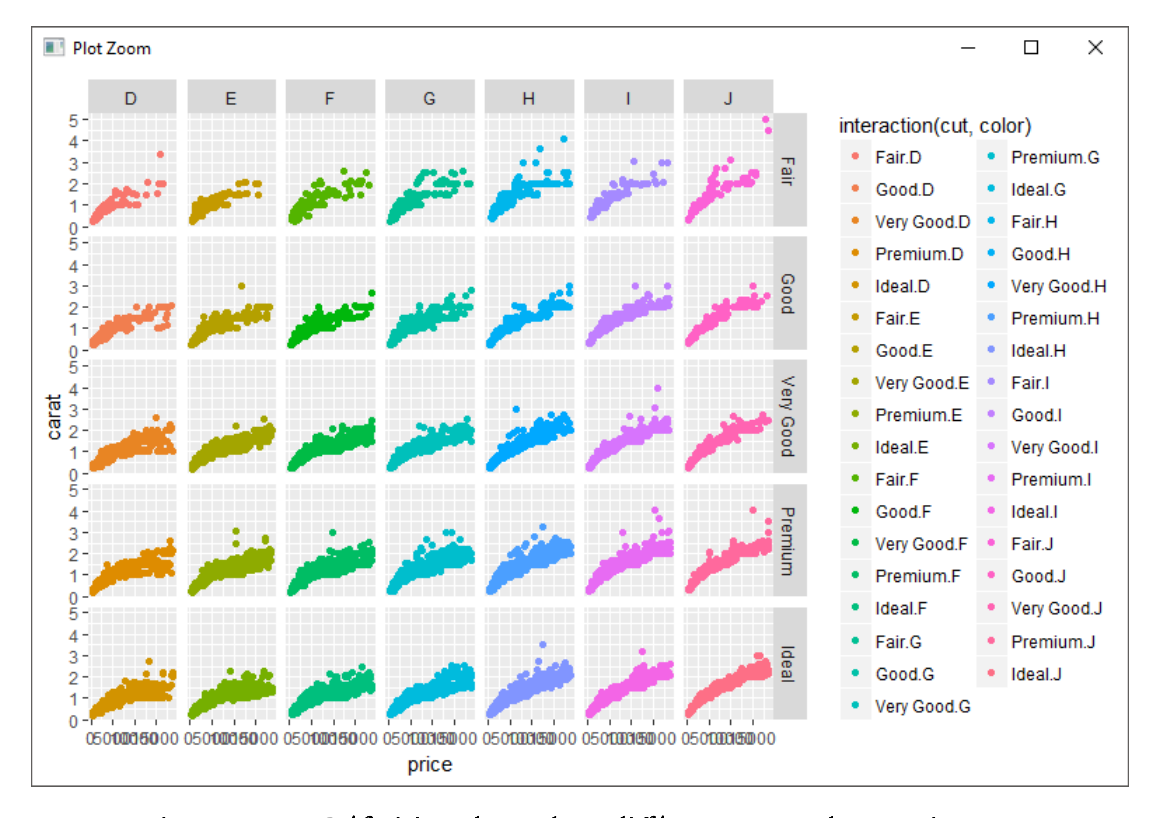


Figure 1.19 – Définition de couleur différente pour chaque vignette

2

La fonction ggplot() et la grammaire graphique

Contents

Nous pouvons nous contenter d'utiliser **qplot()**, précédemment étudiée, cependant on a depuis constaté que l'on est limité avec les options ou paramétrages graphiques (légendes, axes, la pagination...). Ce n'est sans doute pas pour ça que **ggplot2** est au top des packages R, les plus téléchargés. En effet, **gg** dans **ggplot()** est l'abréviation de **G**rammar of **G**raphics l'oeuvre de **Leland WILKINSON** sur les composants graphiques essentiels. La **grammaire graphique** décrit la méthodologie de conception d'un graphique statistique composants (ou couches ou layers) par composants (ou couches ou layers). Ainsi, la vraie puissance de ce package réside dans sa grammaire ou mieux dans les composants ou couches ou layers qui, nous permettent un haut niveau de paramétrage afin de concevoir de véritables œuvres graphiques. Dans ce chapitre, nous allons étudier les composants et découvrir les différentes couches de la grammaire graphique en les illustrant pas à pas avec la fonction **ggplot()** qui en est la base. Enfin nous verrons les principaux types de graphiques en mettant en avant tout le plein potentiel de **ggplot2**.

2.1 Les composants de la grammaire graphique ggplot2

Comme introduit, une visualisation graphique ggplot se construit à partir d'un ensemble de couches ou composants indépendants. Ces couches constituent la grammaire de la syntaxe. Les principales couches de la grammaire sont :

- **Les données brutes ou Data** : le jeu de données contenant les variables que l'on veut visualiser.
- **Les Aesthetiques** (aes): désignation des variables à représenter, en incluant également les autres paramètres Aesthetiques précédemment étudiés à savoir couleurs, les tailles, formes ...
- **Les objets géométriques** (geom...) : qui décident du type de graphique à projeter
- **Les transformations statistiques** (stat...) : éventuellement des transformations peuvent être opérées sur les données.
- **Les échelles ou Scales** (scale...) : permet de contrôler le lien entre les données et les paramètres Aesthetiques (modification de couleurs,gestion des axes...).
- **Le facettage** : nous avons abordé plus haut permet de mettre en évidence des sous-ensemble ou groupe dans les données
- **Le système de coordonnées**

2.1.1 Données (data) et le paramétrage aesthétique ou le "Comment ?"

Tout graphique ggplot2 obéit à une même structure de base. En premier lieu, on appelle la fonction ggplot() en lui passant en paramètre le jeu de données (obligatoirement un data.frame) au minimum ou en ajoutant les caractères aesthétiques. Donc, pour créer un objet **p** de type ggplot il faut au moins l'une des structures suivantes (on peut varier la syntaxe):

```
> #(1) un objet vide
> p <- ggplot()
> #(2) un objet avec seulement le jeu de données
> p <- ggplot(data = )
> #(3) un objet avec mapping données et Aesthetiques
> p <- ggplot(data = , mapping = aes(x = , y =, fill = ,
   colour =,...))
> #(4) un objet + une couche d'aesthetique
> # (resultat identique à la ligne de dessus )
> p <- ggplot(data = ) + aes(x = , y =, fill = , colour
   =,...)
```

ggplot2 utilise les caractères aesthétiques pour définir les propriétés visuelles du graphique, à savoir la variable **x** en abscisse, celui des **y** en ordonnée, la couleur (**colour**), celle de remplissage (**fill**), le type de lignes (**linetype**), ... Une construction graphique est avant tout un mapping entre un jeu de données et les paramètres aesthétiques qui lui donne son aspect visuel du moins virtuellement (car à ce niveau il n'y a aucun moyen de visualiser le graphique). Ce

mapping passe par la fonction **aes()** (qui vit à l'intérieur d'autre fonction) qui indique la correspondance entre les variables d'intérêt du jeu de données et les aesthétiques du graphique. Voyons ce que peut donner les fonctions d'analyse des structures de données R sur un objet **ggplot2** :

```
> p <- ggplot(data = diamonds, aes(x = carat, y = price,
           colour = color, fill = color))
> class(p)
[1] "gg"     "ggplot"
> str(p)
List of 9
 $ data       :Classes 'tbl_df', 'tbl' and 'data.frame':
           53940 obs. of  10 variables:
  ..$ carat  : num [1:53940] 0.23 0.21 0.23 0.29 0. ...
  ..$ cut    : Ord.factor w/ 5 levels "Fair"<"Good"...
  ..$ color  : Ord.factor w/ 7 levels "D"<"E"<"F"< ...
  ..$ clarity: Ord.factor w/ 8 levels "I1"<"SI2" ...
  ..$ depth  : num [1:53940] 61.5 59.8 56.9 62.4  ...
  ..$ table  : num [1:53940] 55 61 65 58 58 57 57 ...
  ..$ price  : int [1:53940] 326 326 327 334 335  ...
  ..$ x      : num [1:53940] 3.95 3.89 4.05 4.2   ...
  ..$ y      : num [1:53940] 3.98 3.84 4.07 4.23  ...
  ..$ z      : num [1:53940] 2.43 2.31 2.31 2.63  ...
  ...
  ...
 $ mapping   :List of 4
  ..$ x      : symbol carat
  ..$ y      : symbol price
  ..$ colour: symbol color
  ..$ fill   : symbol color
  ...
  ...
$ labels     :List of 4
  ..$ x      : chr "carat"
  ..$ y      : chr "price"
  ..$ colour: chr "color"
  ..$ fill   : chr "color"
 - attr(*, "class")= chr [1:2] "gg" "ggplot"
```

Comme on peut le voir dans la structure de notre objet, seul le mapping a été réalisé, tel que nous l'avons définit et à ce niveau si l'on tente de projeter le visuellement (**print**(p) ou **plot**(p)...) le graphique, ce dernier produit une erreur, car nous n'avons pas encore définit l'objet graphiques à tracer. Nous allons voir progressivement comment ajouter les couches ou layers ou composants à notre objet initialisé.

2.1.2 Couches ou layers d'objets géométriques ou le " quoi ?"

Pour ajouter une couches ou layers à notre objet, on utilise l'opérateur d'addition **+**. En ajoutant une couche additionnelle, nous joignons le "comment ?"(aesthetiques) au "quoi ?"

à l'objet graphique que nous définition dans la nouvelle couche. Les layers ou couches, définis grâce à la fonction layer(...) et son paramètre geom ou directement avec la fonction geom_xxx() (où "xxx" représente le type d'objet graphique("point", "histogram" ...)), permettent d'ajouter le type d'objet graphique souhaité. La grammaire à ce niveau peut donc prendre les formes suivantes :

```
> # (1) si p a fait l'objet de mapping déjà
> p <- p + geom_xxx(...)
> # ou
> p <- p + layer(geom =, ... )
> # (2) sinon dans le cas d'un objet ggplot à créer
> p <- ggplot(data = ,mapping = aes(x =,y =,colour = ,...)) +
        geom_xxx(...)
> # ou
> p <- ggplot(data = ,mapping = aes(x=,y=,colour =,...)) +
        layer(geom =,...)
```

Par ailleurs, Chaque fonction geom_xxx() admet également des arguments particuliers permettant de modifier le graphique comme on peut le voir dans le tableau 2.1 (couleur, taille de points, épaisseur de traits, etc.).

Table 2.1 – Liste de quelques fonctions geom_xxx()

geom_xxx()	Paramètres
geom_blank()	aucun paramètre (sert à faire des graphes vides)
geom_abline()	slope, intercept, size, linetype, colour, alpha
geom_hline()	y, intercept, size, linetype, colour, alpha
geom_vline()	x, intercept, size, linetype, colour, alpha
geom_text()	x, y, label, size, colour, alpha, hjust, vjust, parse
geom_point()	x, y, size, shape, colour, fill, alpha, na.rm
geom_jitter()	x, y, size, shape, colour, fill, alpha, na.rm
geom_segment()	x, xend, y, yend, size, linetype, colour, alpha, arrow
geom_line()	group, x, y, size, linetype, colour, alpha, arrow
geom_bar()	x, y, size, linetype, colour, fill, alpha, weight()
geom_histogram()	x, y, size, linetype, colour, fill, alpha, weight()
geom_area()	group, x, y, size, linetype, colour, fill, alpha, na.rm
geom_ribbon()	group, x, ymin, ymax, size, linetype, colour, fill, alpha, na.rm
geom_linerange()	x, ymin, ymax, size, linetype, colour, alpha
geom_pointrange()	x, y, ymin, ymax, size, shape, linetype, colour, fill, alpha
geom_errorbar()	x, ymin, ymax, size, linetype, colour, alpha, width
geom_errorbarh()	x, xmin, xmax, y, size, linetype, colour, alpha, height
geom_crossbar()	x, y, ymin, ymax, size, linetype, colour, fill, alpha, width, fatten
geom_boxplot()	x, ymin, lower, middle, upper, ymax, size, colour, fill, alpha, weight(), width(), outliers(), outlier.size, outlier.shape, outlier.colour
geom_path()	group, x, y, size, linetype, colour, alpha, na.rm, arrow, linemitre, linejoin, lineend
geom_polygon()	group, x, y, size, linetype, colour, fill, alpha

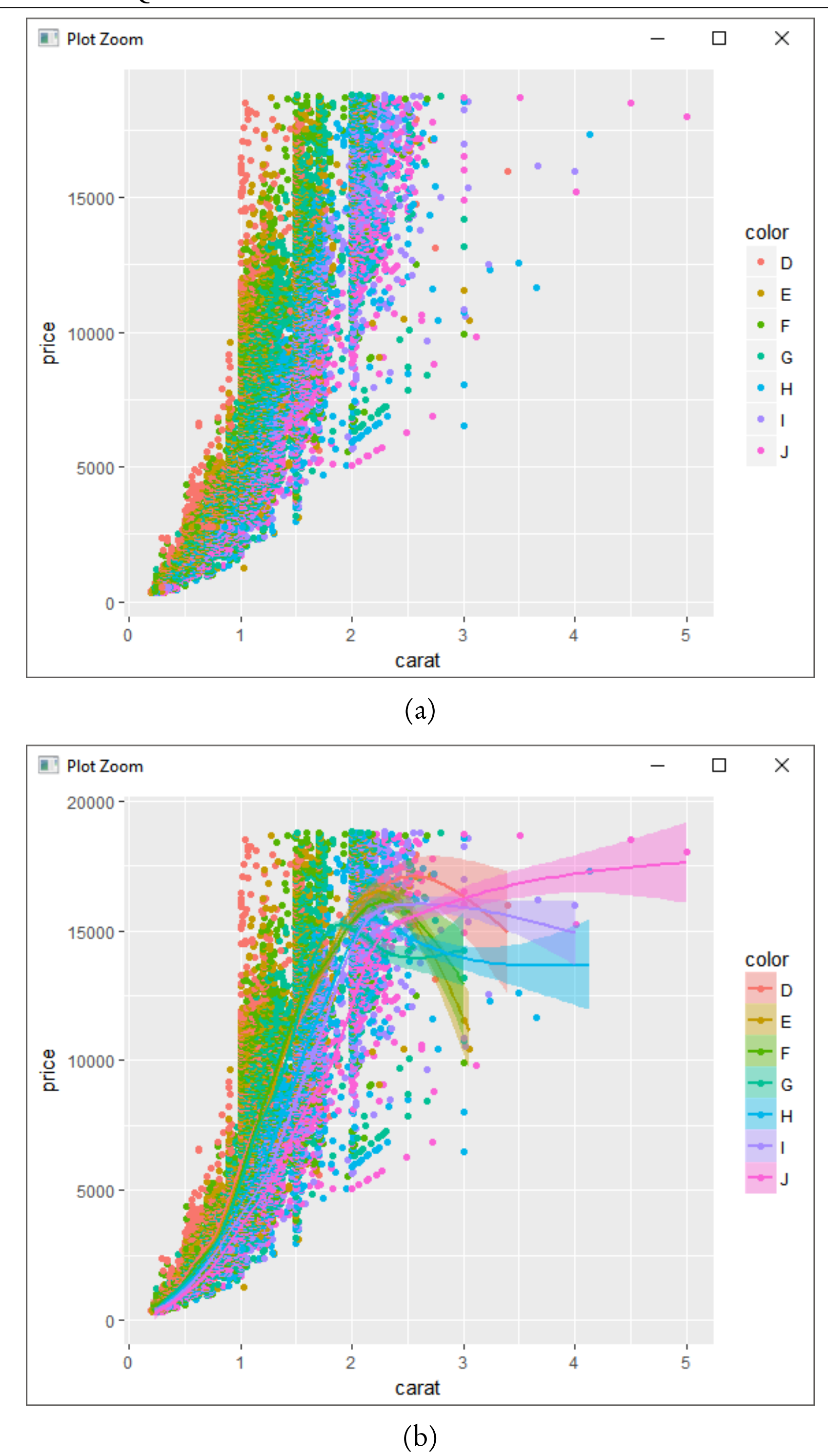
Table 2.1 – Liste de quelques fonctions geom_xxx()

geom_xxx()	Paramètres
geom_rect()	xmin, xmax, ymin, ymax, size, linetype, colour, fill, alpha
geom_rug()	x, y, size, linetype, colour, alpha
geom_step()	group, x, y, size, linetype, colour, alpha, direction
geom_bin2d()	xmin, xmax, ymin, ymax, size, linetype, colour, fill, alpha, weight()
geom_tile()	x, y, size, linetype, colour, fill, alpha
geom_hex()	x, y, size, colour, fill, alpha
geom_density()	group, x, y, size, linetype, colour, fill, alpha, weight(?)
geom_density2d()	group, x, y, size, linetype, colour, alpha, weight(), na.rm, arrow, linemitre, linejoin, lineend
geom_contour()	group, x, y, size, linetype, colour, alpha, weight(?), na.rm, arrow, linemitre, linejoin, lineend
geom_freqpoly()	group, x, y, size, linetype, colour, alpha, weight(?)
geom_quantile()	group, x, y, size, linetype, colour, alpha, na.rm, arrow, linemitre, linejoin, lineend
geom_smooth()	group, x, y, size, linetype, colour, alpha, weight, quantiles, formula, xseq, method, na.rm, arrow, linemitre, linejoin, lineend

Ainsi, pour illustration si nous voulons visualiser notre objet **p**, précédemment initialiser sous forme de nuage de point nous allons ajouter la couche **geom_point()** :

```
> p <- p + geom_point()
> # pour afficher
> plot(p) # ou
> print(p) # ou
> p
> # on peut ajouter la courbe d'ajustement
> p <- p + geom_smooth()
> plot(p)
> # on aurait pu écrire le tout comme ceci
> p <- ggplot(data = diamonds, aes(x = carat, y = price,
              colour = color, fill = color)) +
              geom_point() + geom_smooth()
```

Du code ci-dessus, il en résulte les visuels suivants 2.1. Comme on peut le voir, les objets géométriques ont hérité des aesthétiques précédemment définis.



(a)

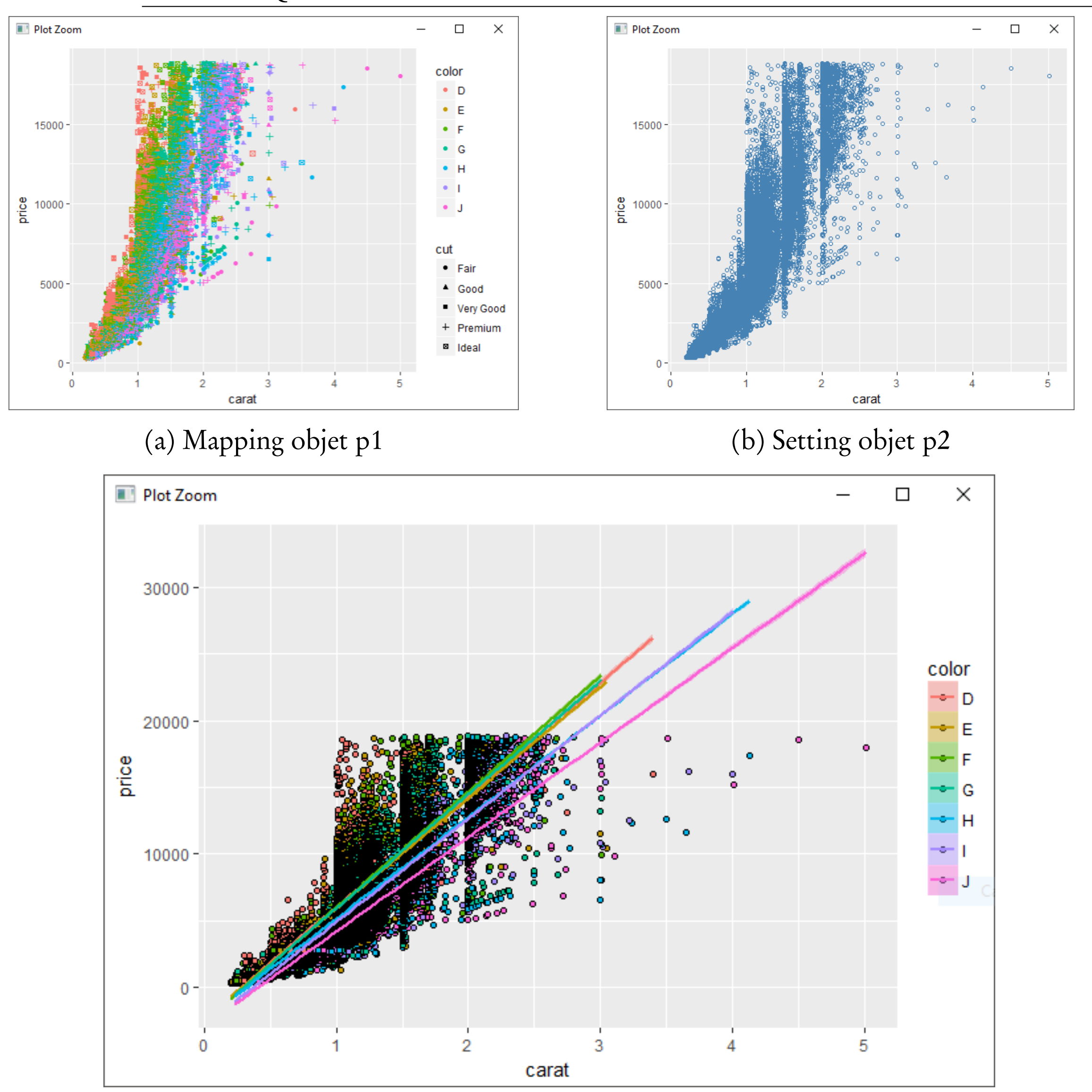
(b)

Figure 2.1 – Application de couche géométrique à un objet ggplot

Définition des paramètres aesthétiques dans la fonction geom_xxx()

Lorsque les paramètres aesthétiques sont définis dans la fonction geom_xxx() qui convient, si la valeur du paramètre dépend d'une variable dans le jeu de données alors il est nécessaire de le définir dans la fonction aes() et placer cette dernière dans geom_xxx() : c'est le **mapping**. Sinon si l'on doit spécifier manuellement le paramètre il faut le définir comme constant directement dans la fonction geom_xxx() correspondant : c'est le **setting**.

```
> ### mapping vs setting
> p <- ggplot(data = diamonds, aes(x = carat, y = price))
> # mapping
> p1 <- p + geom_point(aes(shape = cut,colour =color))
> # setting
> p2 <- p + geom_point(shape = 21,colour = "steelblue")
> plot(p1); plot(p2)
> ### mapping + setting
> ggplot(data = diamonds,
     # mapping d'aesthetiques
     aes(x = carat,y = price,colour = color,fill = color)) +
     # setting d'aesthetiques avec geom_point
     geom_point (colour = 'black', shape = 21) +
     geom_smooth(method="lm")
> # ou on peut adopter l'écriture suivante
> # equivalente
> ggplot(data = diamonds ) +
     # mapping d'aesthetiques
     aes(x = carat,y = price,colour = color,fill = color) +
     # setting d'aesthetiques avec geom_point
     geom_point(colour ='black', shape=21) +
     geom_smooth(method="lm")
```



(a) Mapping objet p1

(b) Setting objet p2

(c) combinaison de Mapping et Setting d'aesthetiques

Figure 2.2 – Définition des paramètres Mapping vs Setting

Aussi, l'on peut avoir déjà définit certains paramètres au niveau de ggplot() et quand même les altérer en les supprimant ou modifiant comme pour l'objet p3 et p4 du code ci-dessus et dont voici le visuel à la figure 2.3:

```
> p <- ggplot(data = diamonds, aes(x = carat, y = price,
           colour = color, fill = color)) + geom_point()
> # suppression de couleur, du remplissage
> # (avec se = FALSE pour omettre l'intervalle de confiance)
> p3 <- p + geom_point(aes(colour = NULL, fill = NULL)) +
         geom_smooth(method="lm", se = FALSE)
> # ajout de forme en mapping et de taille en setting
> p4 <- p + geom_point(aes(shape= cut), size = 3) +
         geom_smooth(method="lm", se = FALSE)
> plot(p3); plot(p4)
> # lintetype modifie le type de ligne
```

```
> ggplot (data = diamonds ) +
  # mapping d'aesthetiques
  aes(x = carat , y = price , colour=color ,
      fill=color , linetype = color) +
  # setting d'aesthetiques avec geom_point
  geom_point(color ='black',shape=21) +
  geom_smooth()
```

(a) suppression objet p3

(b) Ajout objet p4

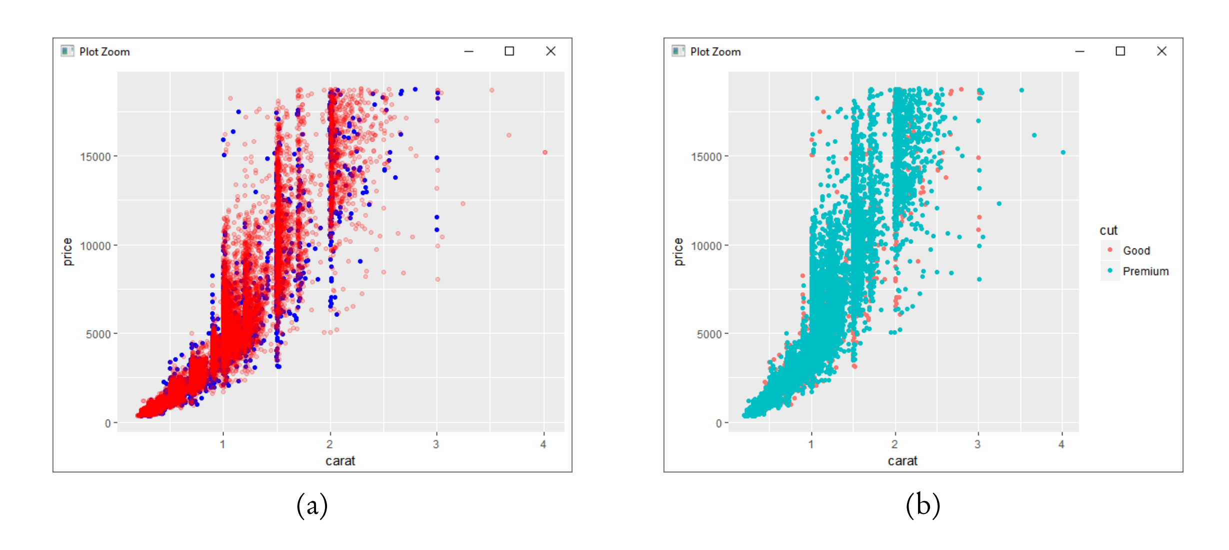
(c) Ajout de lignes différentes

Figure 2.3 – Définition des paramètres dans la fonction geom_xxx()

Les jeux de données par couches

Les fonctions geom_xxx() possèdent des paramètres optionnels. Si les paramètres sont omis, les valeurs automatiquement attribuées sont héritées de celles de ggplot(), il s'agit de notamment data, aes() . . . Ainsi, les couches géométriques peuvent posséder des données autres que celles définies dans ggplot() :

```
> good <- subset(diamonds,cut=="Good")
> premium <- subset(diamonds,cut =="Premium")
> # exemple de geom_xxx() avec paramètres explicités
> p1 <- ggplot(data = good, aes(x=carat, y=price)) +
       geom_jitter (colour="blue") +
       geom_jitter(data = premium, colour="red", alpha= 0.2)
> # exemple de geom_xxx() avec paramètres hérités
> gp <- rbind(good, premium)
> p2 <- ggplot(data=gp,aes(x=carat,y=price,colour = cut )) +
              geom_jitter()
> plot(p1);plot(p2)
```



(a) (b)

Figure 2.4 – Des couches ou layers avec des jeux de données différents

2.1.3 Scales ou Echelles

Les fonctions de type scale_xxx() permettent de contrôler le mapping entre les données et les caractères aesthetiques. Ainsi, Chaque paramètre easthétique possède son échelle et sa fonction scale_xxx(). On peut décomposer les échelles en 4 catégories que nous allons définir et traiter plus loin en détails :

- échelles de positionnement,
- échelles de couleur,
- échelles manuelles discrètes,
- échelles à l'identique ou identity(pas de mise à l'échelle).

Logiquement, les échelles sont automatiquement créées lorsque l'on a déjà définit le mapping entre données et aesthetiques. Cependant, on peut apporter plus de contrôle sur ces aesthetiques pour plus de finesse en ajoutant une couche ou layer avec une fonction **scale_xxx()**. Ce qui modifie les palettes de couleur, gère les axes, les tailles... Les fonctions scales ou d'échelles sont souvent nommées avec les 2 ou 3 composants suivants :

- un préfixe **scale_**
- un radicale le nom de l'aesthétique (**color_**, **fill_**,**shape**,**x_**, **y_**)
- un suffixe qui est le nom du scale : **manual**, **identity**...

Exemple: pour changer l'échelle qui gère la couleur, en présence d'une variable continue dont dépend la couleur, on utilisera **scale_colour_gradient()**. Un exemple visuel est présenté plus bas.
La liste des fonctions d'échelles se trouve dans le tableau 2.2:

Table 2.2 – Les fonctions scales ou échelles

ESTHÉTIQUE	VARIABLE DISCRÈTE	VARIABLE CONTINUE
Transparence (alpha)	scale_alpha_discrete() scale_alpha_manual() scale_alpha_identity()	scale_alpha_continuous() scale_alpha_identity()
Couleur (colour)	scale_colour_discrete() scale_colour_brewer() scale_colour_grey() scale_colour_hue() scale_colour_manual() scale_colour_identity()	scale_colour_continuous() scale_colour_dilstiller() scale_colour_gradient() scale_colour_gradient2() scale_colour_gradientn() scale_colour_identity()
Remplissage (fill)	scale_fill_discrete() scale_fill_brewer() scale_fill_grey() scale_fill_hue() scale_fill_manual() scale_fill_identity()	scale_fill_continuous() scale_fill_distiller() scale_fill_gradient() scale_fill_gradient2() scale_fill_gradientn() scale_fill_identity()
Type de ligne (linetype)	scale_linetype_discrete() scale_linetype_manual() scale_linetype_identity()	scale_linetype_continuous() scale_linetype_identity()
Forme(shape)	scale_shape_discrete() scale_shape_manual() scale_shape_identity()	scale_shape_continuous() scale_shape_identity()
Taille (size)	scale_size_discrete() scale_size_manual() scale_size_identity()	scale_size_continuous() scale_size_area() scale_size_identity()
Position (x, y)	scale_x_discrete() scale_y_discrete()	scale_x_continuous() scale_y_continuous() scale_x_date() scale_y_date() scale_x_datetime() scale_y_datetime()

Table 2.2 – Les fonctions scales ou échelles

ESTHÉTIQUE	VARIABLE DISCRÈTE	VARIABLE CONTINUE
		scale_x_log10() scale_y_log10() scale_x_reverse() scale_y_reverse() scale_x_sqrt() scale_y_sqrt()

La majorité de ces fonctions partagent quelques paramètres en commun, il s'agit notamment de:

- **name**: permet d'annoter les axes ou la légende. Mais au lieu de recourir à une fonction scales pour annoter les axes, nous avons les fonctions spécifiques comme **xlab()** pour l'axe des abscisses, **ylab()** pour les ordonnées, **labs(**title=,x=, y=) qui permet d'annoter les deux axes en même temps et **ggtitle()** pour ajouter un titre. On peut utiliser des expressions mathématiques pour annoter les axes pour plus de détails **?plotmath**.
- **limits**: prend un vecteur contenant le minimum et le maximum. On a également des fonctions spécifiques pour gérer les limites telles que **lims(x=,y=)**, **xlim()**, **ylim()**
- **breaks** et **labels**: le premier définit les graduations et le second étiquette ces dernières.
- **na.value**: permet de spécifier la valeur par défaut en cas de présence de valeur manquante **NA**.

Bien évidement nous verrons juste ci-dessous ou bien plus tard dans la galerie graphique un exemple de l'utilisation de ces paramètres.

Les échelles de positionnement

Les fonctions d'échelles de positionnement permettent de contrôler les axes notamment. On a notamment d'une part **scale_x_continuous()** et **scale_y_continuous()** qui permet de gérer les axes des variables continues et d'autres part on a **scale_x_discrete()** et **scale_y_discrete()** pour les variables discrètes. Pour les différentes transformations, les fonctions d'échelles disposent d'un paramètre très intéressant nommé **trans** qui prend en valeur les éléments de la première colonne du tableau 2.5 ci-dessous :

Name	Function $f(x)$	Inverse $f^{-1}(y)$
asn	$\tanh^{-1}(x)$	$\tanh(y)$
exp	e^x	$\log(y)$
identity	x	y
log	$\log(x)$	e^y
log10	$\log_{10}(x)$	10^y
log2	$\log_2(x)$	2^y
logit	$\log(\frac{x}{1-x})$	$\frac{1}{1+e(y)}$
pow10	10^x	$\log_{10}(y)$
probit	$\Phi(x)$	$\Phi^{-1}(y)$
reciprocal	x^{-1}	y^{-1}
reverse	$-x$	$-y$
sqrt	$x^{1/2}$	y^2

Figure 2.5 – Les différentes transformations

Il y a également quelques fonctions d'échelles qui sont disponibles pour effectuer certaines transformations à savoir scale_x_log10(), scale_y_log10(), et scale_x_sqrt()...

```
> # log transformation de x, y
> p <- ggplot(diamonds,aes(carat,price)) + geom_point()
> p +  scale_x_continuous(name = "logarithme des carats",
      trans = "log10") +
     scale_y_continuous(name = "logarithme des prix",
      trans = "log10")
> # ou
> p + xlab("logarithme des carats") +
    ylab("logarithme des prix") +
    scale_x_log10() + scale_y_log10()
```

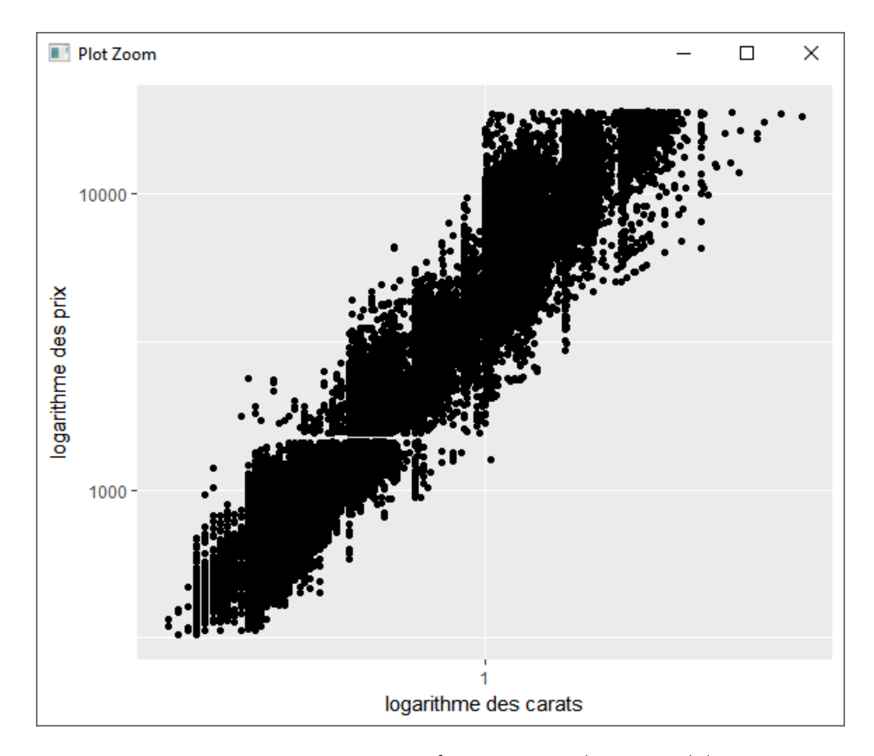


Figure 2.6 – Log-transformation des variables

```
> p <- ggplot(diamonds,aes(x = clarity,fill = cut )) + geom_
  bar()
> # spécification de graduation discrète avec limites
> # et renommage de catégories
> p + scale_x_discrete(limits = c("I1", "SI2", "SI1"),
                   labels = c("Coup_1","Coup_2","Coup_3"))+
    scale_y_continuous(breaks = c(0,500,5000,10000,14000),
      labels = c(0,500,5000,10000,14000),limits = c(0,15000))
```

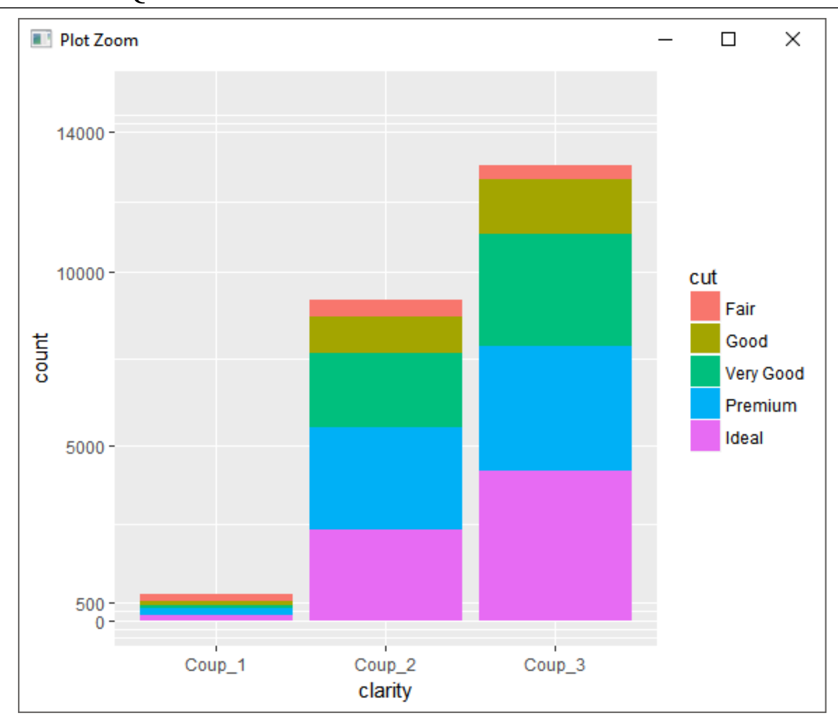


Figure 2.7 – Définition des graduations

```
> # réduire l'espacement (voir explication plus bas)
> p + scale_y_continuous(expand = c(0,0)) +
      scale_x_discrete(expand = c(0,0))
```

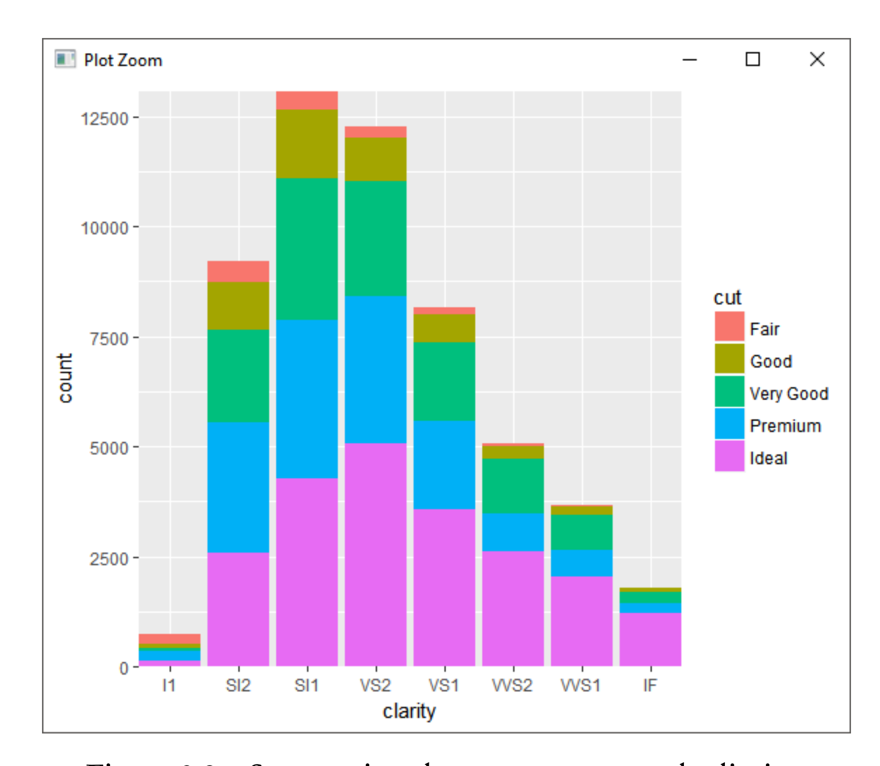


Figure 2.8 – Suppression des espaces au tour des limites

```
> # reverser l'ordre de y
> p + scale_y_reverse()
```

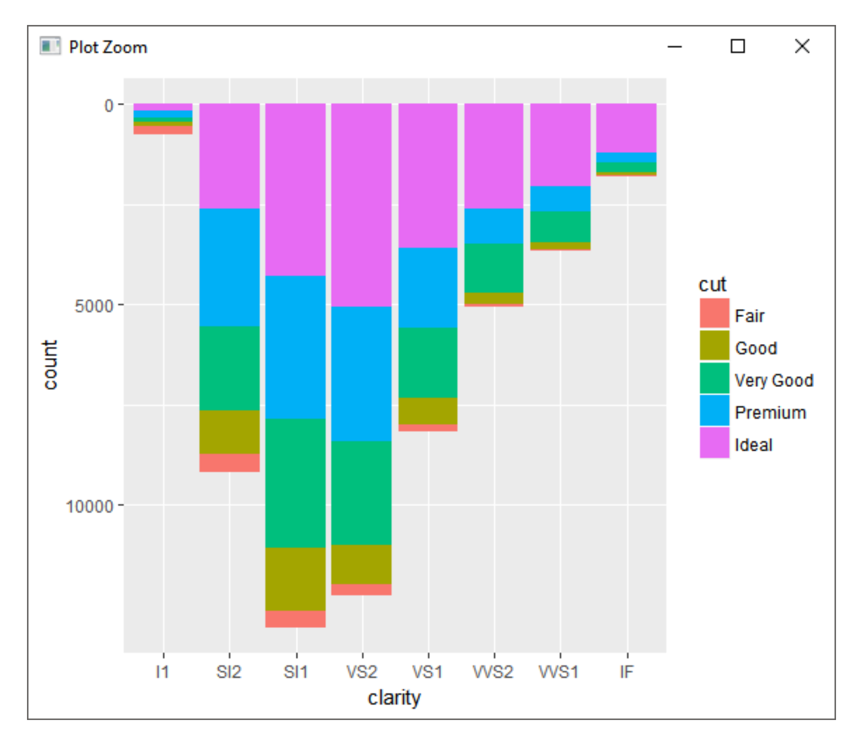


Figure 2.9 – Renversement du repère

Nous pouvons également, modifier l'ordre ou définir l'ordre des différents niveaux de la variable catégorielle en abscisse avec le paramètre **limits** :

```
> # définir l'ordre des barres
> p + scale_x_discrete(limits = c("VVS2", "VVS1", "IF",
                  "I1","SI2","SI1","VS2","VS1"))
```

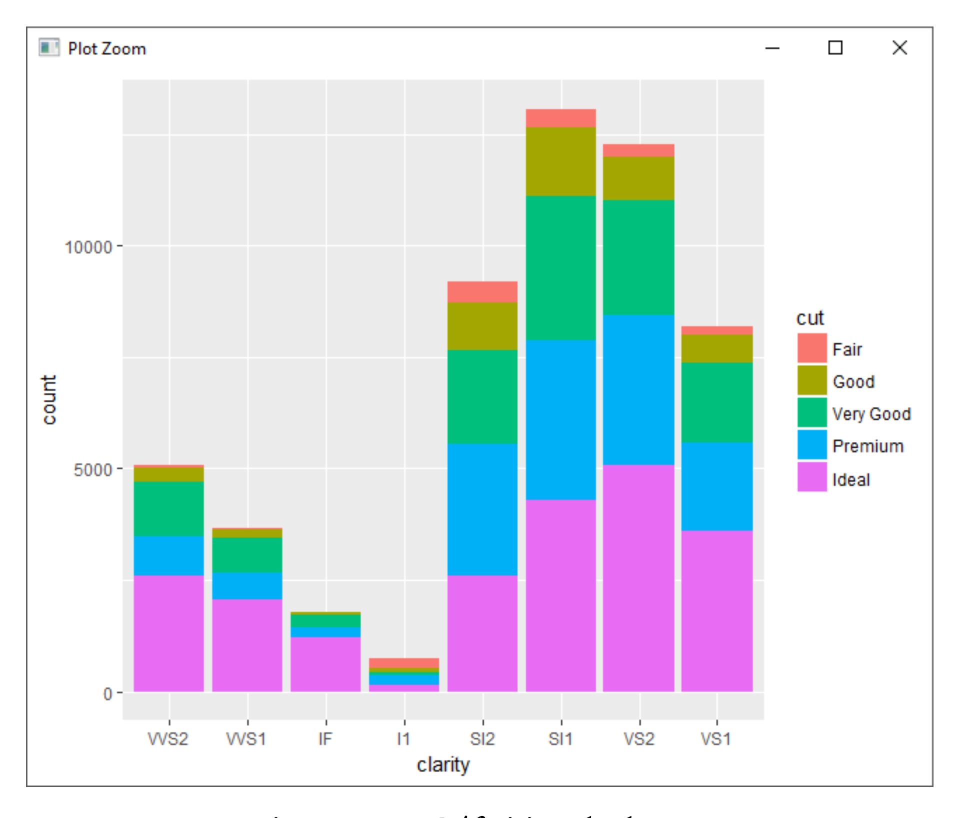


Figure 2.10 – Définition des barres

Et l'ordre de remplissage à l'intérieur des barres en redéfinissant les niveaux de la variable catégorielle fournit à **fill** :

```
> # Définir l'ordre à l'intérieur des barres
> ggplot(diamonds,aes(x = clarity,fill = factor(cut,
  levels = c( "Ideal","Good","Very Good",
                                  "Fair","Premium"))))+
  geom_bar()+ scale_fill_discrete("Ordre changé")
```

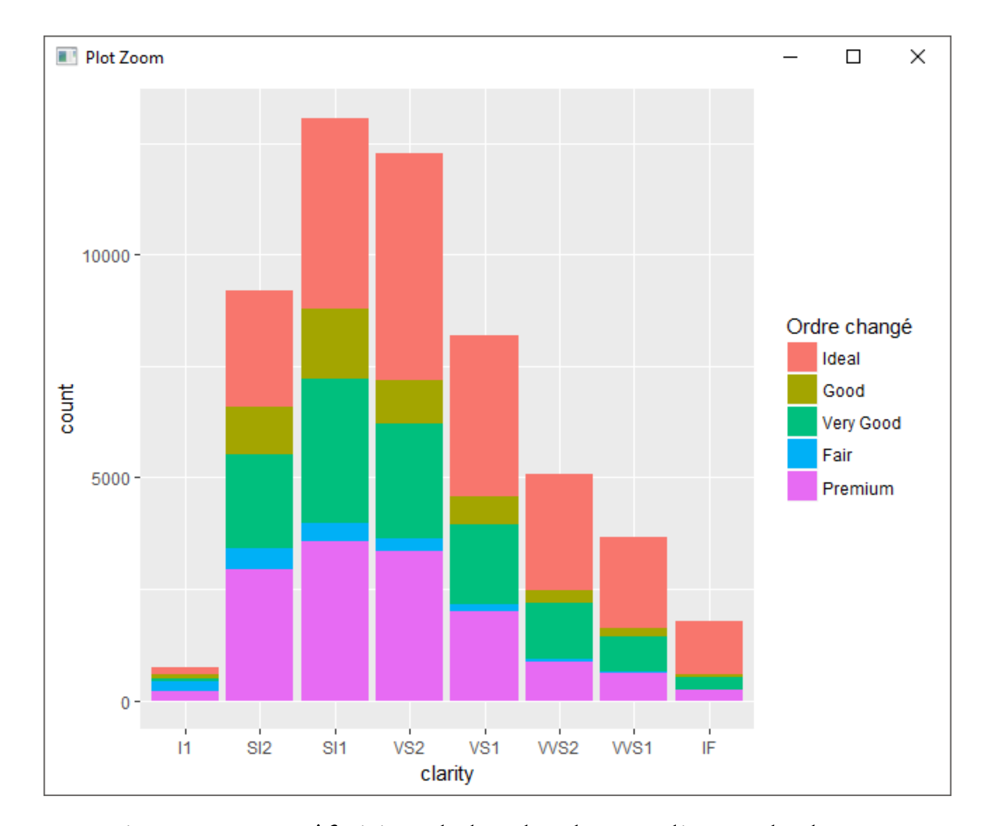
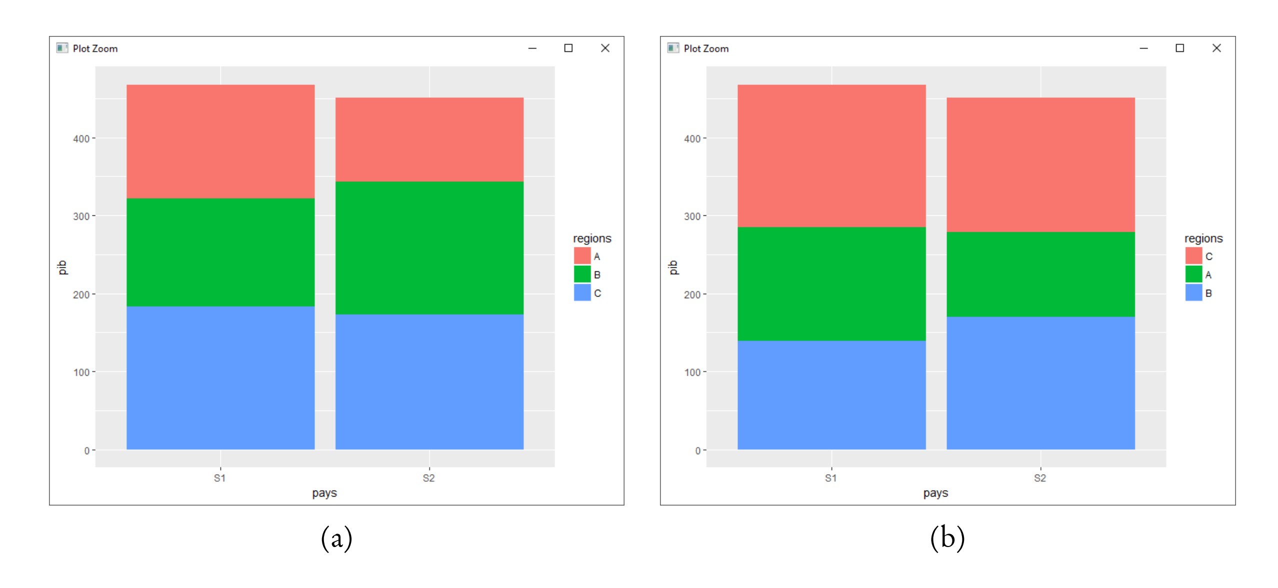


Figure 2.11 – Définition de l'ordre de remplissage des barres

Toutefois, il faut noter que la modification de l'ordre de remplissage des barres est peu trivial lorsque nous utilisons le paramètre **stat**(nous verrons ce paramètre dans section sur les transformations statistiques) et que nous lui affectons la valeur "identity":

```
> data <- data.frame(regions = rep(c("A","B","C"),2),
                     pays = rep(c("S1","S2"),each=3),
                     pib = runif(6,100,250))
> ggplot(data)+aes(x = pays,y = pib, fill = regions)+
  geom_bar(stat = 'identity')
> # rédifinition de l'ordre des niveaux de la variable
   categrielle
> ggplot(data = within(data, regions <-factor(regions,
  levels=c("C","A","B"))))+ aes(x = pays,y = pib, fill =
     regions)+
  geom_bar(stat = 'identity')
```

Pour passer de la figure 2.12a à la figure 2.12b, nous devons impérativement modifier l'ordre des niveaux de la variable catégorielle à l'intérieur du jeu de données comme nous le démontre le code ci-dessus.

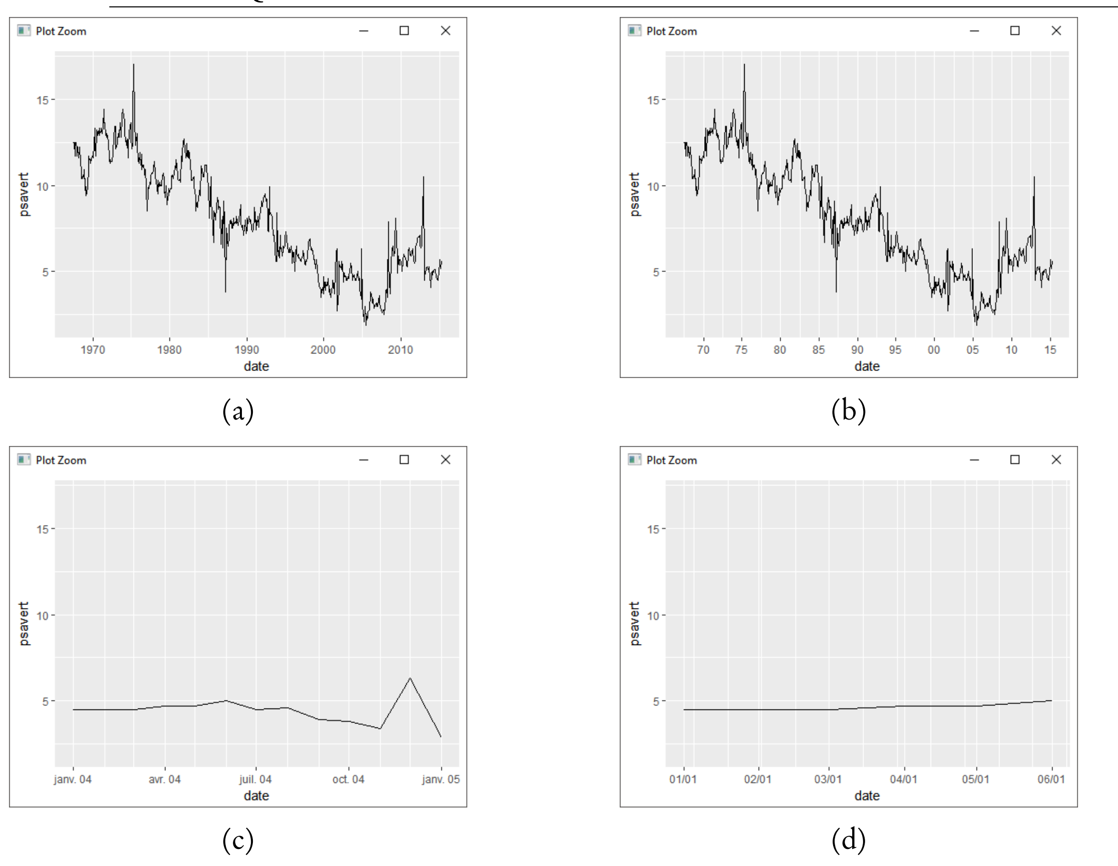


(a) (b)

Figure 2.12 – Modification de l'ordre de remplissage des barres dans le cas de **stat**='identity'

Notons par ailleurs, que même quand nous avons spécifié les limites, **ggplot()** réserve toujours un espace pour éviter que le graphique et les axes ne se superposent pas. On observe à la figure 2.8 que ce n'est pas le cas grâce au paramètre **expand**, l'on peut modifier l'écart entre axe et graphique. ce paramètre prend un vecteur de deux valeurs, le premier est le facteur multiplicateur et le second un facteur additif de l'écart. Ainsi, si l'on fait **expand**=c(0,0), on élimine totalement l'écart. Pour finir cette partie sur les échelles de position, on doit noter le cas particulier des dates qui sont en fait des variables continues avec des étiquettes spéciales. On dispose de **scale_x_date()**,**scale_y_date()**, **scale_x_datetime()**...ou utiliser à la place **scale_x_continuous()**, **scale_y_continuous()** ...mais avec des paramètres spéciaux **date_breaks**/**date_minor_breaks()** et **date_labels** :

```
> data(economics)
> p <- ggplot(economics, aes(date, psavert)) +
           geom_line(na.rm = TRUE)
> plot(p)
> # graduation par 5 ans avec model 1950 ~ 50
> p + scale_x_date(date_labels = "%y",
             date_breaks = "5 years")
> p + scale_x_date(
  limits = as.Date(c("2004-01-01", "2005-01-01")),
  date_labels = "%b %y",
  date_minor_breaks = "1 month")
> p + scale_x_date(
  limits = as.Date(c("2004-01-01", "2004-06-01")),
  date_labels = "%m/%d",
  date_minor_breaks = "2 weeks")
```



(a) (b) (c) (d)

Figure 2.13 – Cas d'échelles de date

La figure 2.14 présente les différents types de lignes et leurs valeurs correspondantes, que prend le paramètre **lignetype**.

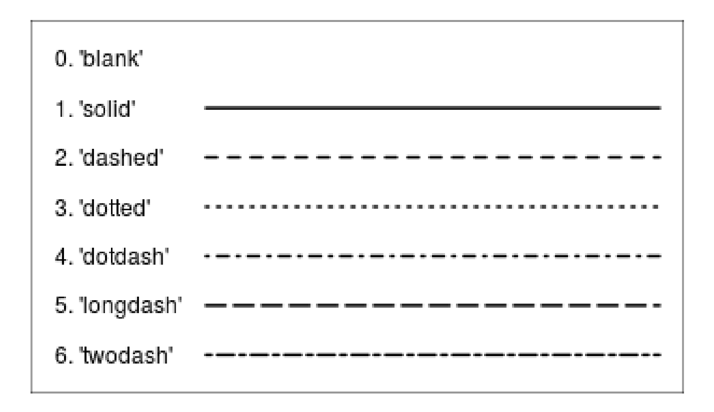


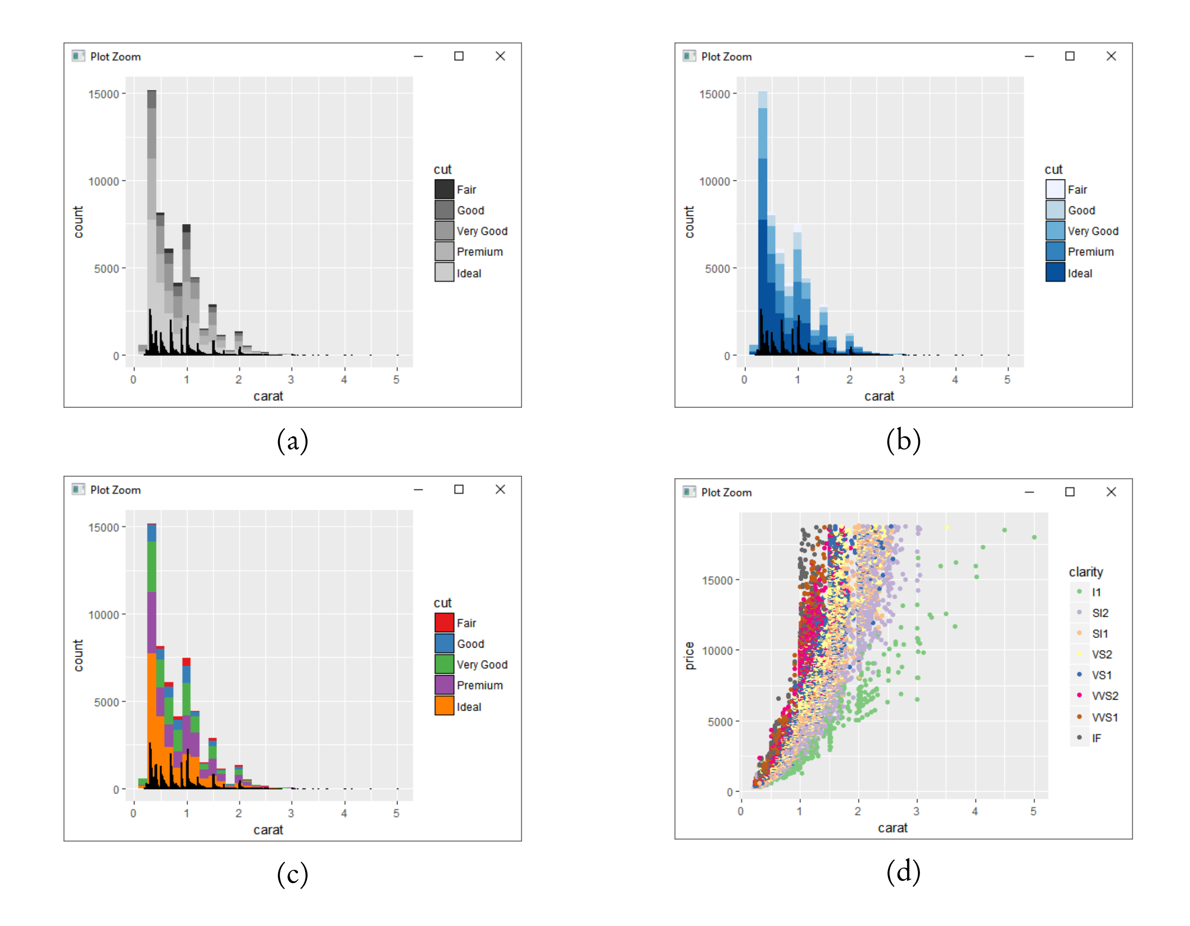
Figure 2.14 – Les différentes types de lignes

Les échelles de couleur (colour et fill)

Pour les couleurs, nous avons les fonctions d'échelles continues et discrètes également que ce soit pour le contour ou le remplissage voir le tableau 2.2.

```
> # échelle de couleur fill & color
> p <- ggplot(diamonds) + aes(carat,fill = cut)+
    geom_histogram()
> p + geom_bar(color="black") +
```

```
    scale_fill_grey()
> p + geom_bar() +
      scale_fill_brewer()
> p + geom_bar(color="black") +
   scale_fill_brewer(palette="Set1")
> # colour nuage de points
> ggplot(diamonds, aes(x = carat, y = price,
   color = clarity)) + geom_point() +
    scale_color_brewer(palette = "Accent")
```



(a) (b) (c) (d)

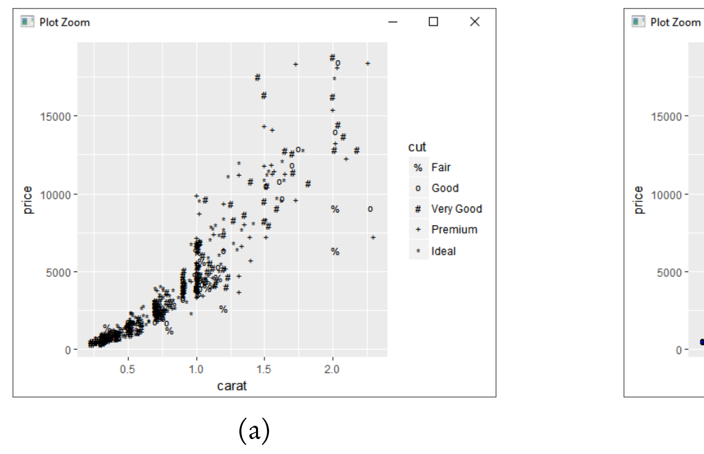
Figure 2.15 – Échelle de couleur

Les échelles manuelles discrètes

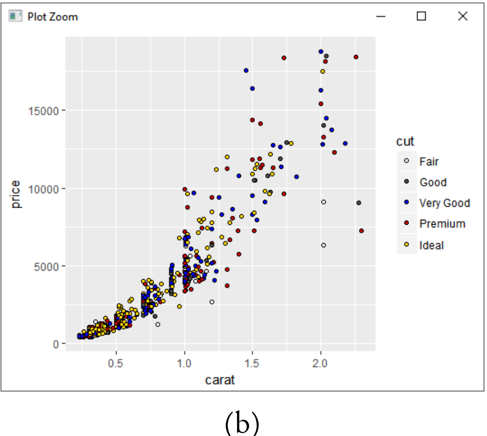
Les échelles manuelles concernent toutes échelles dont les valeurs sont saisies par l'utilisateur lui-même. Il peut s'agir de couleur, forme, taille . . . dont les valeurs sont manuellement saisies.

```
> p <- ggplot(db,aes(carat,price, shape = cut))
> forme <- c("%","o","#","+","*")
> p + geom_point(size=2.5) +
       scale_shape_manual(values = forme)
> # couleur de remplissage avec mapping de
> # classe avec la couleur
> p + geom_point(aes(fill = cut), shape = 21) +
     scale_fill_manual(values = c("Fair"= "white",
```

```
                    "Good" = "gray30" ,
                   "Very Good" = "blue",
                   "Premium"  = "red3",
                    "Ideal"   = "gold"))
```



(a)

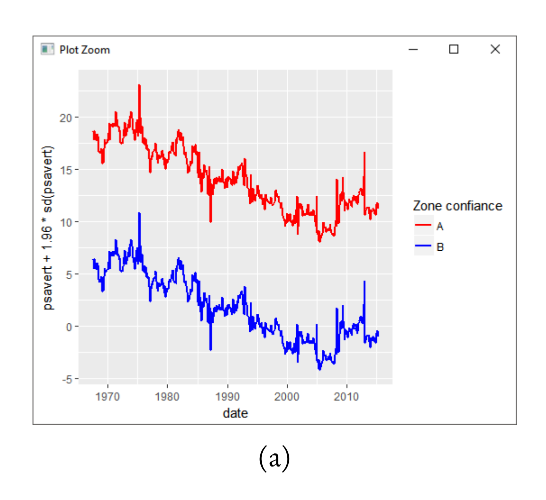


(b)

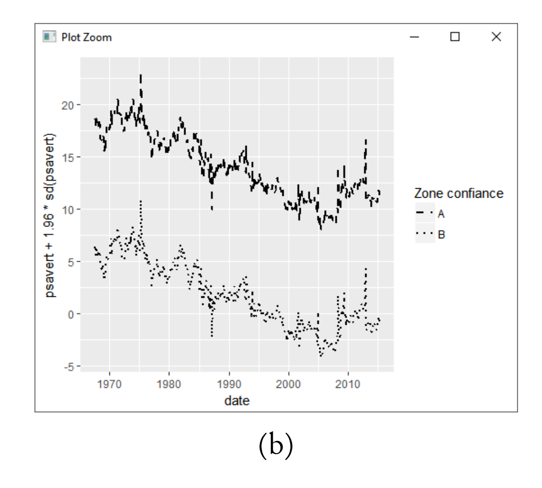
Figure 2.16 – Échelle manuelle de forme et remplissage

Comme on peut le voir dans le code ci-dessous, il est possible d'utiliser une référence pour définir les valeurs d'échelle manuellement. Par exemple, "**max**" attribué d'abord à **colour**, servira à définir dans la fonction échelle manuelle de couleur, la valeur exacte de la couleur.

```
> # utilisation de référence pour la couleur
> p <- ggplot(economics, aes(date))
> p + geom_line(aes(y = psavert + 1.96*sd(psavert),
                  colour = "max"),size=1) +
      geom_line(aes(y = psavert - 1.96*sd(psavert),
                  colour = "min"),size=1) +
      scale_colour_manual(name = "Zone confiance",
                      labels=c("A","B"),
      values = c("max" = "red", "min" = "blue"))
> # utilisation de référence pour le type de ligne
> p + geom_line(aes(y = psavert + 1.96*sd(psavert),
                    linetype = "max"),size=0.9) +
      geom_line(aes(y = psavert - 1.96*sd(psavert),
                    linetype  = "min"),size=0.9) +
      scale_linetype_manual(name = "Zone confiance",
                     labels=c("A","B"),
      values = c("max" = "dashed", "min" = "dotted"))
```



(a)



(b)

Figure 2.17 – Échelle manuelle de couleur et de type de ligne

Les échelles à l'identique

Généralement pour **ggplot2**, les fonctions ou arguments contenant le terme **identity**, ont pour but de respecter les caractéristiques des variables du jeu de données. Ainsi, pour utiliser un jeu de données contenant déjà l'échelle de couleur on fera recours à **scale_colour_identity()**.

```
> val = c(10,20,30,50,35,25,5)
> lab = LETTERS[1:7]
> col = c("violet","blue","seagreen","green",
     "yellow","orange","red")
> ggplot(data = data.frame(lab,val,col)) +
      aes(x = lab, y = val,fill = col) +
      geom_bar(stat = "identity") +
      scale_fill_identity()
> x = c(2,5,7,10);y = c(10,20,25,30)
> col = c("blue","green","orange","red")
> ggplot(data = data.frame(x,y,col)) +
      geom_point(aes(x,y,shape=x,size=y/2,colour = col)) +
      scale_shape_identity() +
      scale_size_identity() +
      scale_colour_identity()
```



Figure 2.18 – Échelle à l'identique

2.1.4 Transformations statistiques

Parfois la réalisation d'un graphique dépend de certaines transformations notamment statistiques opérées sur le jeu de données brutes. Lorsque la transformation est simplement l'application d'une fonction(généralement **log()**. . .), il suffit de spécifier cela directement dans la fonction **aes()** :

```
> ggplot(diamonds) +
      aes(x = log10(carat),y = log10(price)) +
      geom_point() + geom_smooth(method = "lm", se = FALSE)
> df <-  data.frame(x = seq(-10,10, by =0.01))
> f <- function(x){1/(1 + exp(x))}
> ggplot(df) + aes(x = x, y = f(x)) + geom_line()
```

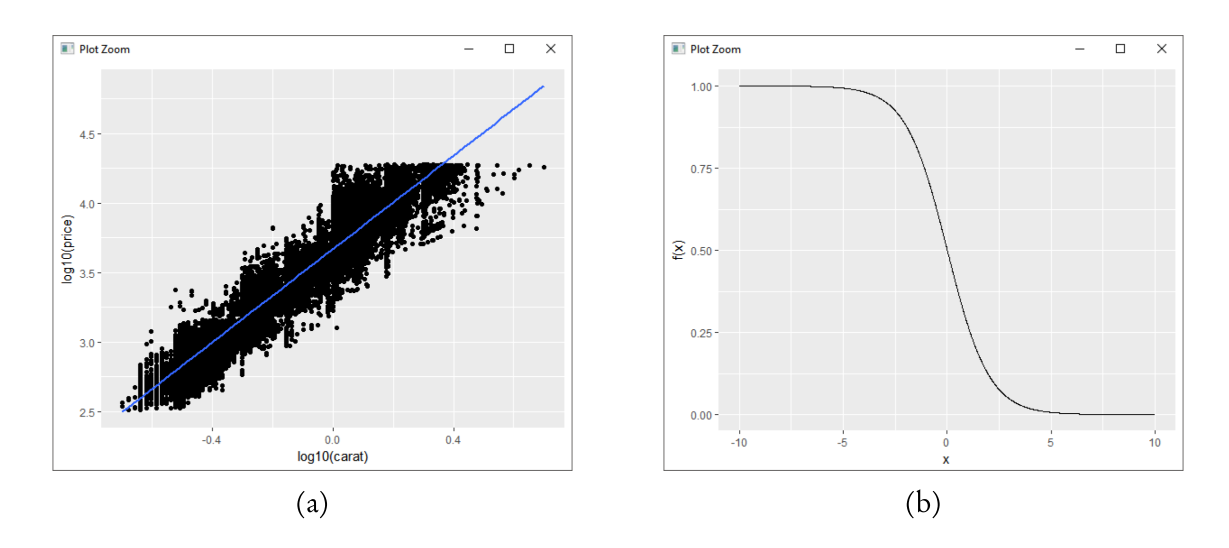
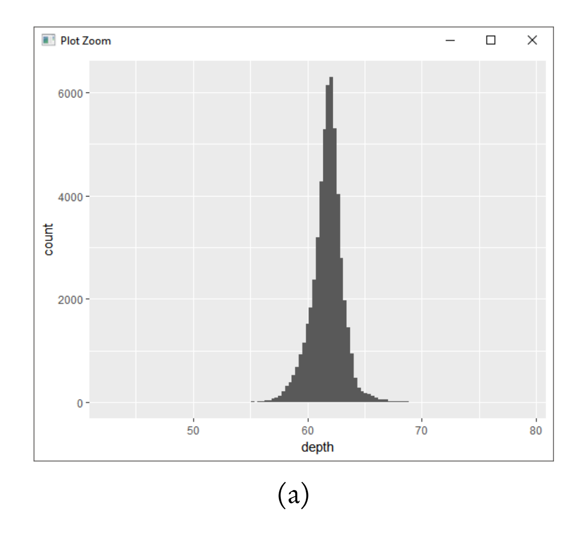


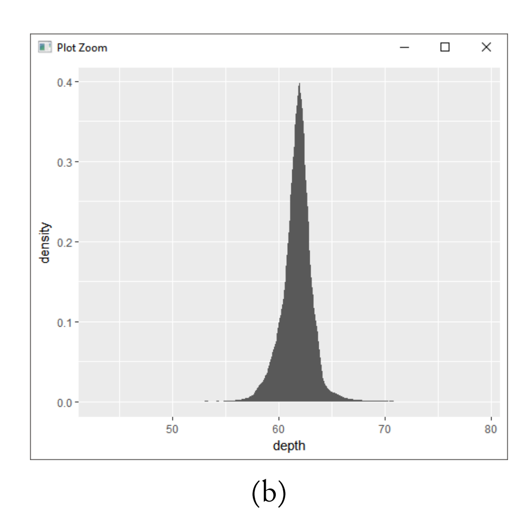
Figure 2.19 – Des transformations fonctionnelles

La réalisation d'autres graphiques nécessite l'utilisation de transformations plus complexes. Pour illustration, l'on peut considérer le cas de l'histogramme ou il faut calculer le nombre d'observations dans chaque classe. Les statistiques permettent de gérer ces éléments intermédiaires dans les graphiques **ggplot**. Ils sont renseignés dans l'argument **stat** des fonctions geom_**xxx()**.

```
> p <- ggplot(diamonds, aes(x = depth))
> # par défaut
> p1 <- p + geom_histogram(stat = "bin",binwidth = 0.3)
> p2 <- p + geom_histogram(stat = "density")
> plot(p1);plot(p2)
```



(a)



(b)

Figure 2.20 – Histogramme avec le paramètre **stat** de la fonction **geom_histogram()**

Mieux, le package **ggplot2** possède une alternative plus flexible pour la transformation de statistique au lieu et de l'argument **stat** plus les fonctions **geom_xxx()**, il s'agit des fonctions **stat_xxx()**. Nous utilisons automatiquement et implicitement certaines de ces fonctions **stat_xxx()**, lorsque nous utilisons certaines fonctions **geom_xxx()** qui elles les invoquent en arrières plan. Comme on peut lire dans le tableau 2.3, à chaque fois que nous utilisons une couche de type **geom_bar()**, **geom_freqpoly()**, **geom_histogram()**, par défaut il y a la fonction **stat_bin()** qui est invoquée en arrière plan pour déterminer les intervalles ou bins :

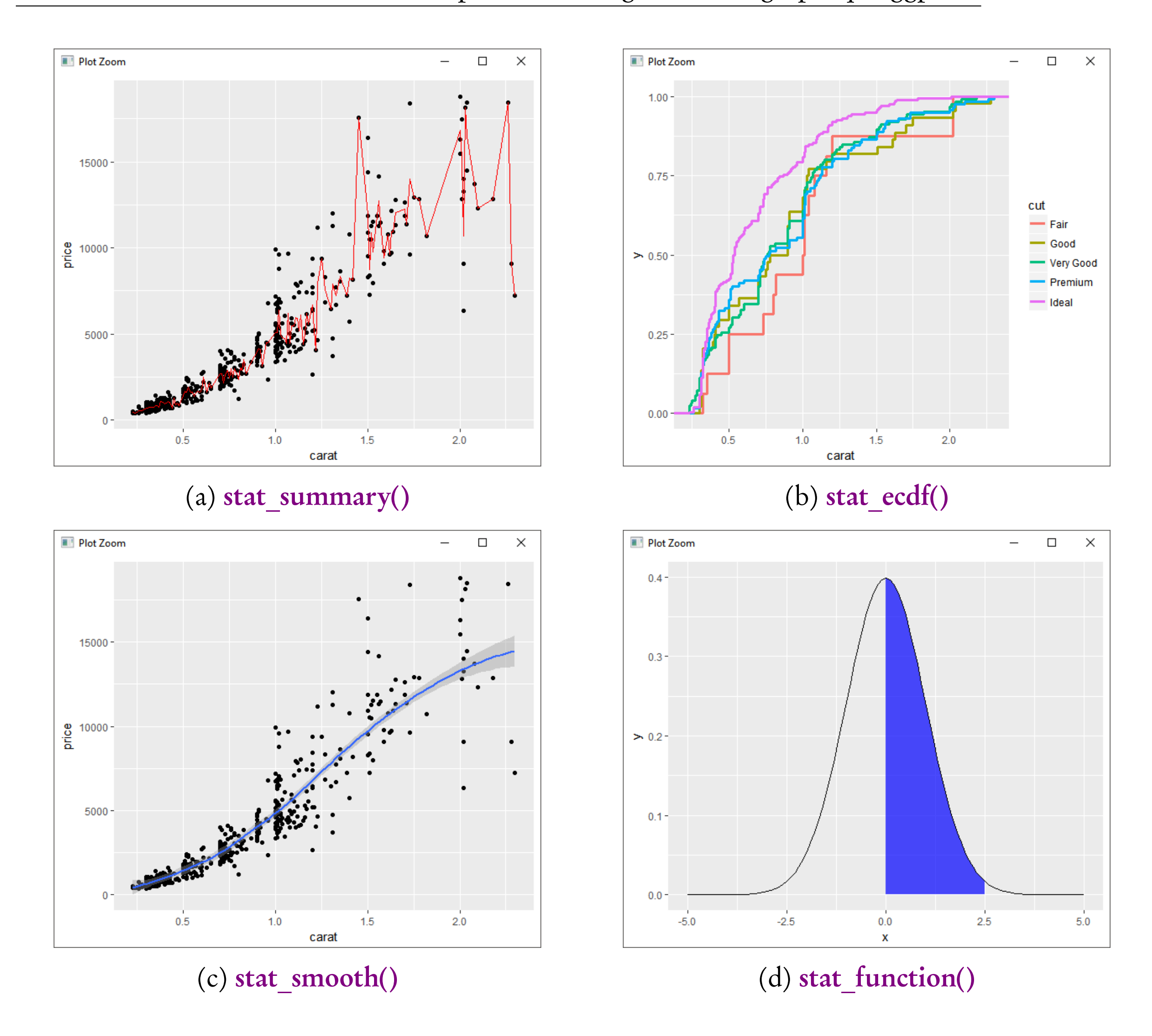
Fonction **stat_xxx()**	Fonctions appelantes **geom_xxx()**
stat_**bin()**	geom_bar(), geom_freqpoly(), geom_histogram()
stat_**bin2d()**	geom_bin2d()
stat_**bindot()**	geom_dotplot()
stat_**binhex()**	geom_hex()
stat_**boxplot()**	geom_boxplot()
stat_**contour()**	geom_contour()
stat_**quantile()**	geom_quantile()
stat_**smooth()**	geom_smooth()
stat_**sum()**	geom_count()

Table 2.3 – Les fonctions statistiques et leurs fonctions géométriques équivalentes

Ainsi, dans le code ci-dessous, les deux premiers objets graphiques sont équivalents pour produire la figure 2.21a, en permettant d'ajouter des statistiques sommaires notamment les moyennes des y en fonction des x (jointes par une ligne):

```
> ### utilisation de stat_xxx() ou geom_xxx()
> ggplot(db, aes(x = carat,y = price)) +
```

```
  geom_point() + # pour tracer le nuage de point
  stat_summary(geom = "line", fun.y = "mean",
     colour = "red", size = 0.7)
> # ou (donne le même graphique donc non reproduit ci-dessus)
> ggplot(db, aes(x = carat,y = price)) +
  geom_point() + # pour tracer le nuage de point
  geom_line(stat = "summary", fun.y = "mean",
    colour = "red", size = 0.7)
> ### fonction de distribution cumulative empirique
> ggplot(db)+ aes(x = carat, colour = cut) +
    stat_ecdf(geom = "step",size = 1.1)
> ### ajouter une courbe d'ajustement stat_smooth()
> ggplot(db) + aes(x =carat,y=price) + geom_point() +
  stat_smooth(method="loess")
> ### stat_function avec la fonction dnorm()
> # créer une fonction pour délimiter la zone
> f <- function(x) {
    y <- dnorm(x)
    y[x < 0 |x > 2.5] <- NA
    return(y)
}
> ggplot(data.frame(x=c(-5,5))) + aes(x = x) +
    stat_function(fun=f, geom="area",
    fill="blue", alpha=0.7) +
    stat_function(fun=dnorm)
```



(a) stat_summary()

(b) stat_ecdf()

(c) stat_smooth()

(d) stat_function()

Figure 2.21 – Illustration de quelques fonctions stat_xxx()

Mais nous avons d'autres fonctions statistiques qui n'ont pas de substitues à travers une fonction géométrique de type geom_xxx()listées dans le tableau 2.4 :

Fonction stat_xxx()	utilités
stat_ecdf()	calcul pour la courbe de distribution cumulée empirique
stat_function()	détermine y en fonction de x
stat_summary()	sommaire statistique de y selon chaque x
stat_summary2d(), stat_summary_hex()	sommaire statistique de données rangées en classes
stat_qq()	opére des calculs pour la tracé un quantile-quantile plot.
stat_spoke()	convertit les angles et radians en position
stat_unique()	supprime les doublons.

Table 2.4 – Les fonctions statistiques sans équivalents géométriques

Les nouvelles variables générées par les transformations statistiques

En effet, les fonctions statistiques ne font pas que transformer mais également créer des variables. En exemple, pour la fonction geom_histogram(), la fonction stat. appelée par défaut

Fonction stat_xxx()	utilité
stat_bin()	répartition des données en classes
stat_contour()	calculer les contours des données en 3d
stat_density()	estimation de densité 1d par la méthode du noyau
stat_density2d()	estimation de densité 2d
stat_identity()	ne transforme pas les données
stat_qq()	qqplot(droite de Henry)
stat_quantile()	quantiles continus
stat_smooth()	lissage
stat_sum()	somme les valeurs uniques
stat_summary()	appliquer une fonction pour faire des summaries sur les valeurs de y
stat_unique()	retire les valeurs dupliquées

Table 2.5 – Les fonctions statistiques les plus utilisées

est stat_bin(). Elle génère les nouvelles variables suivantes :

- **count** : nombre de points dans chaque classe ;
- **density** : densité pour chaque classe ;
- **x** : le centre des classes

Ainsi, nous pouvons utiliser ces nouvelles valeurs au lieu des valeurs contenues dans notre jeu de données originales, en les appelant avec la syntaxe suivante **..***nom_nouvelle_variable***..** :

```
> ggplot(data = diamonds,
        aes(x = depth, y = ..density..)) + geom_histogram() +
        geom_line(stat = "density",col = "red", size = 1)
> # par défaut
> ggplot(diamonds, aes(price, colour = cut)) +
       geom_freqpoly(binwidth = 500)
> # avec la variable ..density.. ( donne donc le même
  graphique)
> ggplot(diamonds, aes(price, colour = cut)) +
  geom_freqpoly(aes(y = ..density..),size=1, binwidth = 500)
> # une densité avec paramètrage de fonction
> ggplot(NULL,aes(x=c(-3,5))) +
  stat_function(fun=dnorm, geom="ribbon",alpha=0.5,
     args = list(mean = 1.5,sd = 1), aes(ymin=0,ymax=..y..))
> # diagramme en barres en histograme avec comme
> # hauteur les proportions
> ggplot(diamonds, aes(x = depth)) +
     stat_bin(binwidth= 1,aes(y = ..count../sum(..count..)))
```

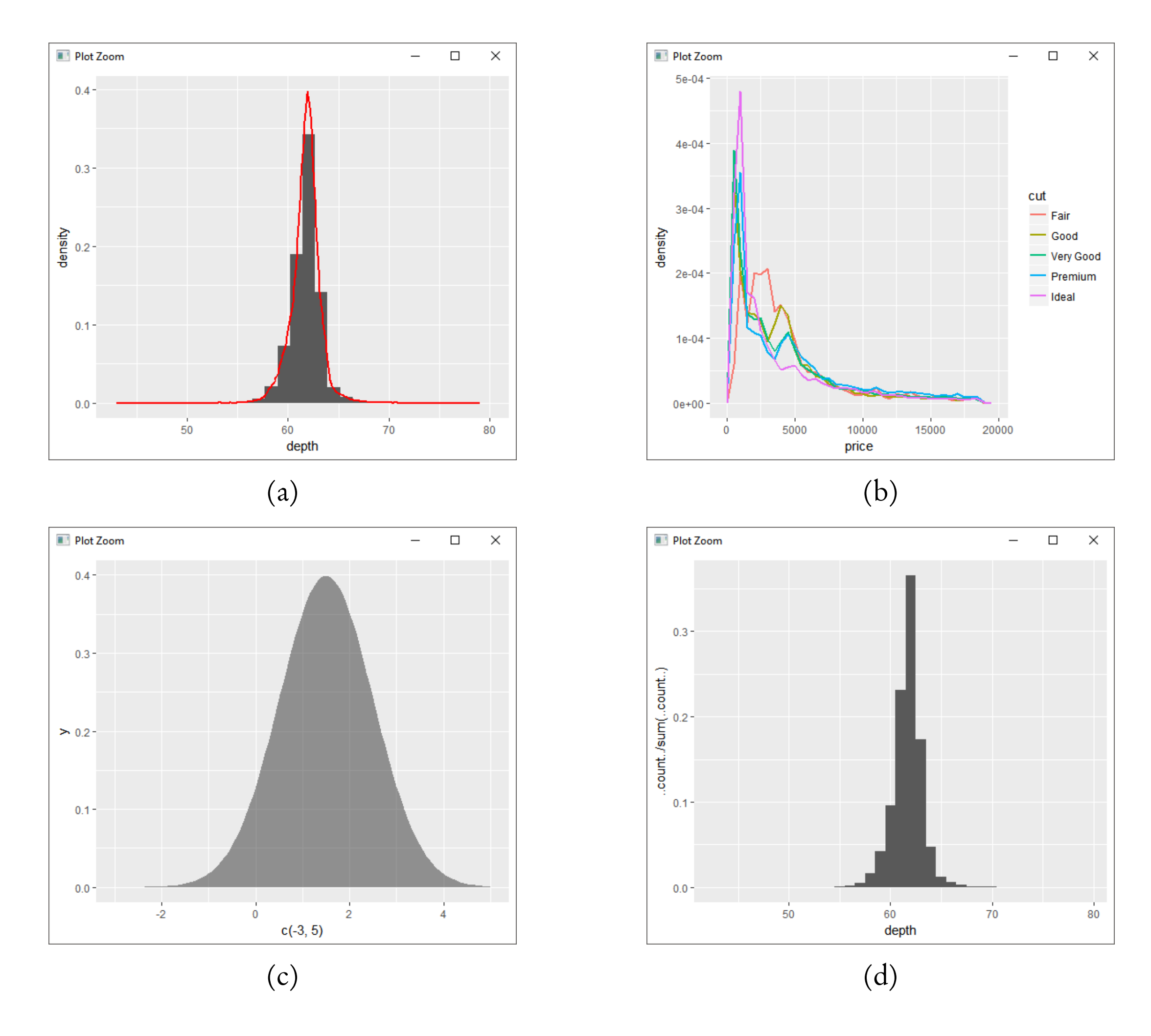
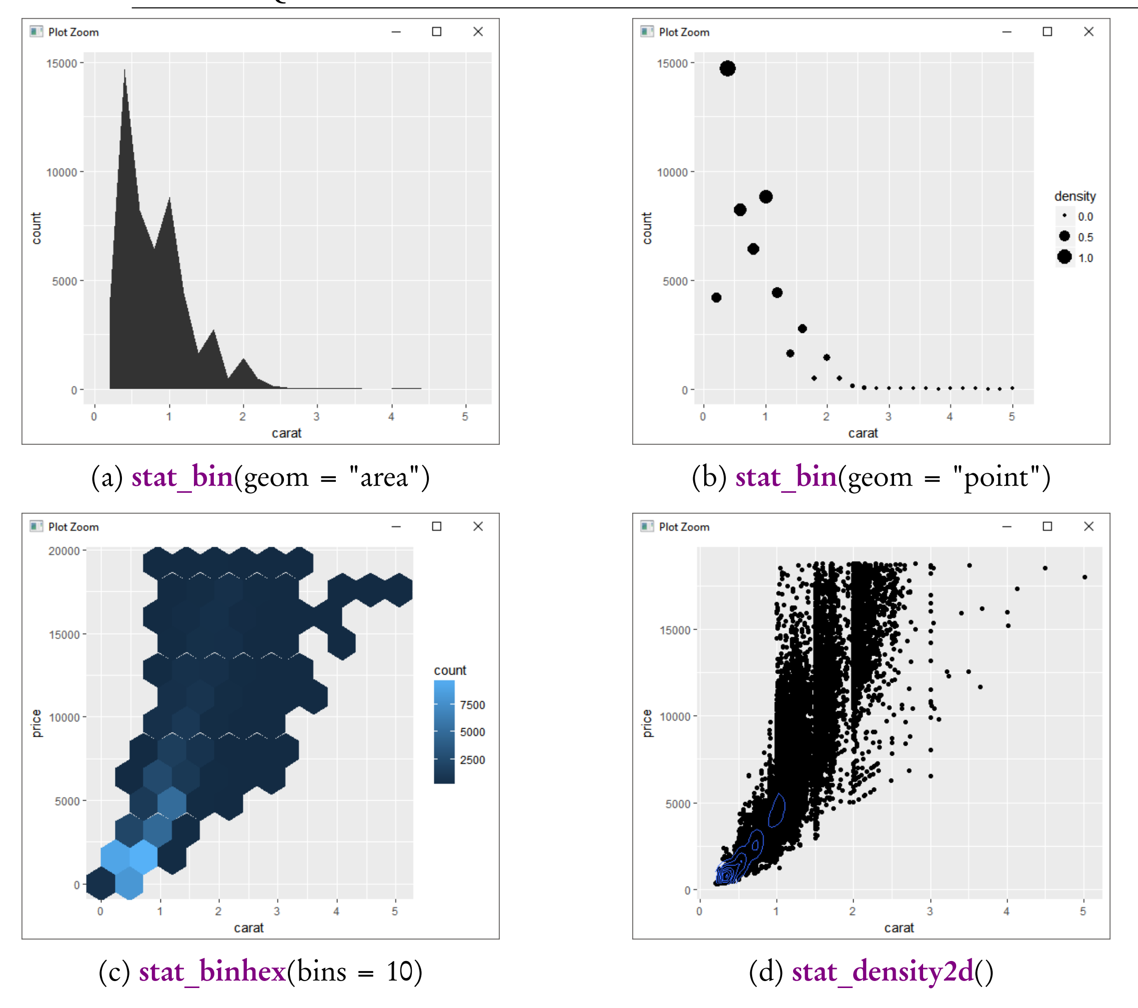


Figure 2.22 – Graphiques réalisés à base de variables générées

Pour obtenir la liste des nouvelles variables calculées pour une fonction statistique, il faut se référer à la documentation. Exemple **?stat_bin** , à la partie **Computed variables**.
Autres illustrations avec les fonctions statistiques :

```
> p <- ggplot(diamonds, aes(carat))
> p + stat_bin(aes(ymax = ..count..),binwidth = 0.2,
         geom = "area")
> p + stat_bin(aes(size = ..density..), binwidth = 0.2,
         geom = "point", position="identity")
> p <- ggplot(diamonds, aes(carat, price))
> p + stat_binhex(bins = 10)
> p + geom_point() + stat_density2d()
```



(a) stat_bin(geom = "area")   (b) stat_bin(geom = "point")

(c) stat_binhex(bins = 10)   (d) stat_density2d()

Figure 2.23 – Autres illustrations avec des fonctions statistiques

Visualiser à l'identique les données

Pour finir, il s'agit parfois d'utiliser des données déjà transformées. C'est l'intérêt de stat_identity() ou du geom_xxx(stat = "identity",...). Dans l'exemple ci-dessous, on ne demande pas à ce que la fonction détermine elle-même le nombre de classe, nos données ont été déjà classées et l'on veut les visualiser telles quelles en barre :

```
> val = c(10,20,30,50,35,25,5)
> lab = LETTERS[1:7]
> ggplot(data = data.frame(lab,val)) + aes(x = lab,y = val) +
       stat_identity(geom = "bar")
> # ou
> ggplot(data = data.frame(lab,val)) + aes(x = lab,y = val) +
      geom_bar(stat = "identity")
```

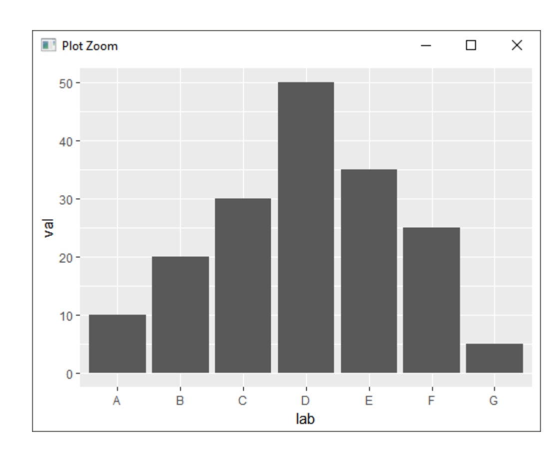


Figure 2.24 – Visualiser à l'identique les données

En effet, lorsqu'on verra une fonction ou paramètre ggplot contenant le terme **identity**, il faut en déduire qu'il s'agit de ne rien transformer et de prendre les données telles que fournies.

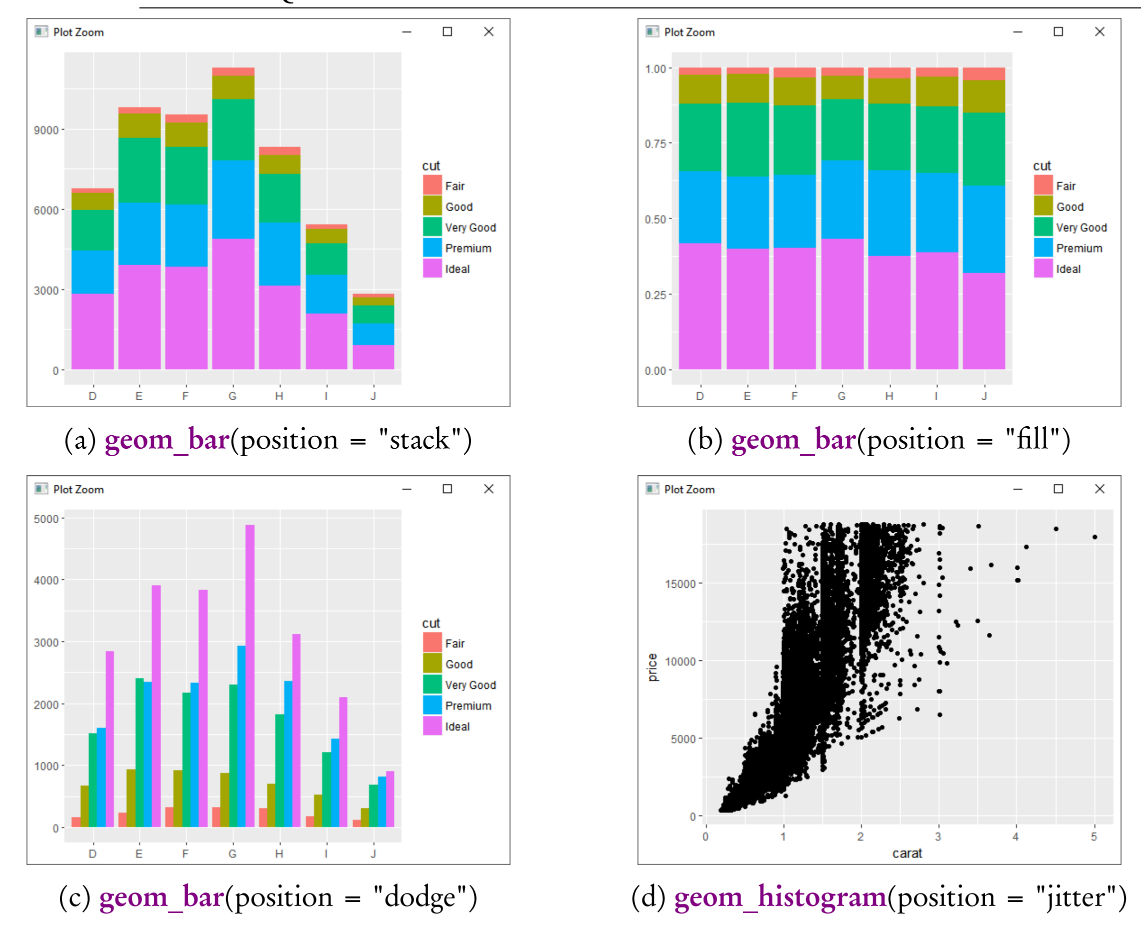
2.1.5 Position des objets géométriques

Les paramètres de position déterminent comment organiser des objets géométriques qui utiliseraient le même espace. Nous avons notamment comme fonctions de position :

- position_dodge(): évite les chevauchements, place les objets côte à côte
- position_stack(): empile les objets ou barres. C'est la position par défaut d'un diagramme en barres
- position_fill(): empile les objets ou barres, avec application d'une échelle.
- position_jitter(): place les objets géométriques côte à côte en essayant d'optimiser l'espace

On a également position_identity(), qui ne modifie rien et peut s'avérer pertinent certains type de graphique. On peut appeler ces fonctions également en combinant le paramètre position avec les fonctions geom_xxx() :

```
> p <- ggplot(diamonds, aes(color, fill = cut)) +
        xlab(NULL) + ylab(NULL)
> p + geom_bar() # stack par défaut
> p + geom_bar(position = "fill")
> p + geom_bar(position = "dodge")
> ggplot(diamonds) +
      geom_point(aes(carat, price),
                 position = "jitter")
```



(a) geom_bar(position = "stack")   (b) geom_bar(position = "fill")

(c) geom_bar(position = "dodge")   (d) geom_histogram(position = "jitter")

Figure 2.25 – Ajustement de la position des objets

2.1.6 Système de coordonnées

Le système de coordonnées permet de contrôler le plan où l'on repère les objets graphiques. Selon le plan adopter, l'objet géométrique peut adopter un comportement différent. Jusqu'ici, le plan est basé sur un repère linéaire de type cartésien. On distingue notamment :

- coord_cartesian() : coordonnées cartésiennes, par défaut
- coord_fixed() : coordonnées cartésiennes avec la même échelle pour les deux axes,
- coord_flip() : coordonnées cartésiennes avec les axes renversés,
- coord_map() : projections pour les cartes,
- coord_polar() : coordonnées polaires,
- coord_trans() : coordonnées cartésiennes transformées.

Les figures 2.28 et 2.26, présentent quelques transformations possibles du système de coordonnées.

```
> p <- ggplot(data = diamonds) +
  geom_bar(aes(x = factor(1), fill = cut), width = 1)
> p # par défaut cartésien
> # utiliser "y" pour définir les angles theta
```

```
> p + coord_polar(theta = "y")
> p + coord_flip()
> ggplot(data = diamonds) +
  geom_bar(aes(x = cut, fill = cut), width = 1) +
    coord_polar()
```

(a) coord_cartesian()

(b) coord_polar(theta = "y")

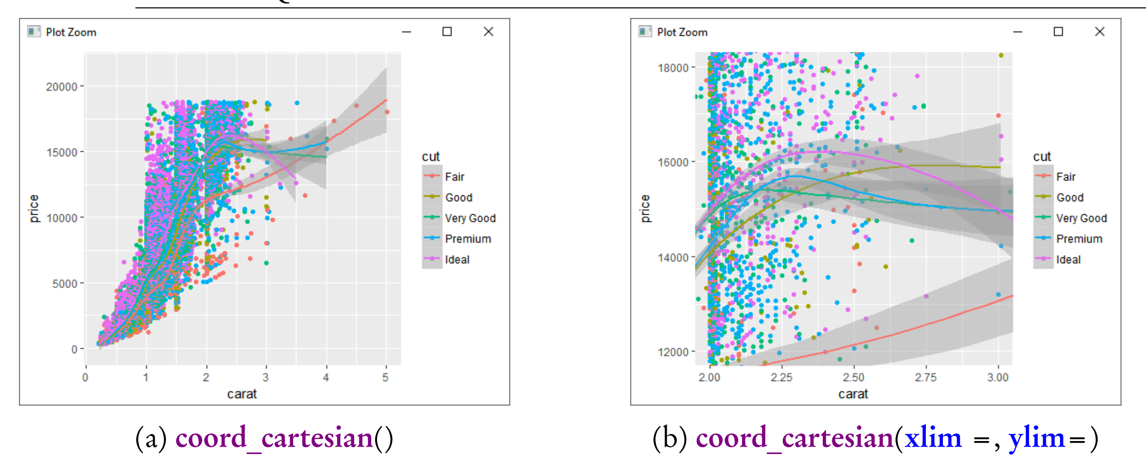
(c) coord_flip()

(d) coord_polar(xlim =, ylim=)

Figure 2.26 – Quelques systèmes de coordonnées 1

Par ailleurs, nous parlerons plus en détails des graphiques en secteur à la section dédiée au niveau de la galerie.
Le code ci-dessous présente un effet zoom en utilisant un système de coordonnée réduit, l'idée étant de se focaliser à une partie spécifique :

```
> p <- ggplot(diamonds)+aes(carat,price,colour = cut)+
  geom_point() + geom_smooth()
> p
> # zoom une partie spécifique
> p + coord_cartesian(xlim = c(2, 3),ylim = c(-12000,18000))
```

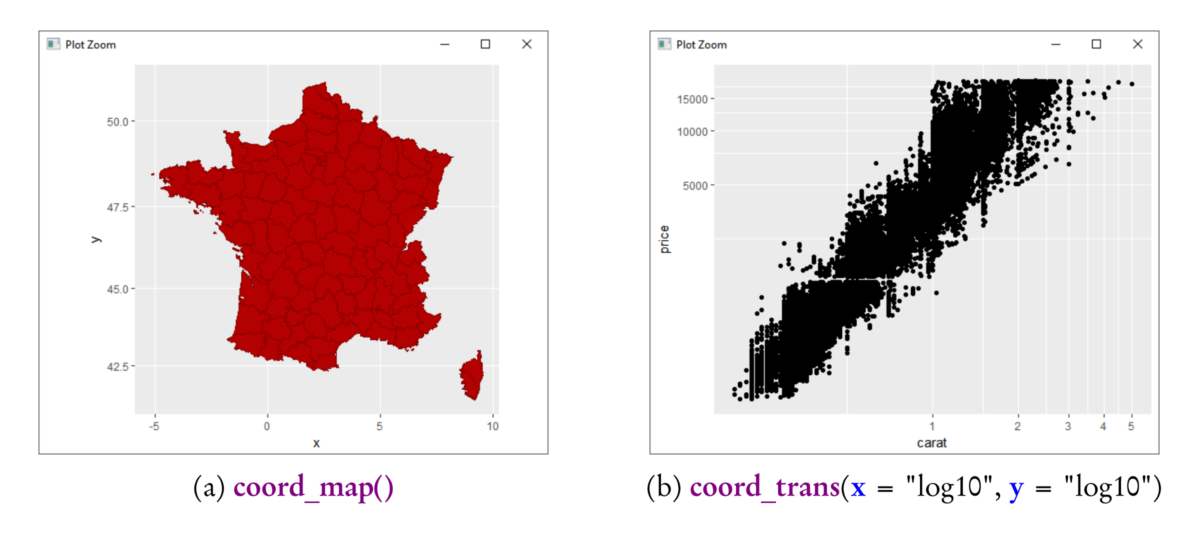


(a) coord_cartesian()

(b) coord_cartesian(xlim =, ylim=)

Figure 2.27 – Quelques systèmes de coordonnées 2

```
> # visualiser une carte
> library(maps)
> # obtenir les données grâce au package maps
> fr = data.frame(map(database="france", plot=F)[c("x", "y")
  ])
> # utiliser le système de coodonnée géographique
> ggplot() + coord_map() +
    geom_polygon(data = fr,aes(x=x, y=y),fill=hsv(0, 1, 0.7),
    color=hsv(0, 1, 0.5), size=0.2)
> ## 3 manières pour une log-transformation
> # (1) en transformant les variables aesthetiques
> ggplot(diamonds,aes(log10(carat),log10(price))) +
    geom_point()
> # (2) avec scales ou échelles
> ggplot(diamonds,aes(carat,price,log="xy")) + geom_point() +
        scale_x_log10() + scale_y_log10()
> # (3) changer le système de coordonnées
> ggplot(diamonds,aes(carat,price)) + geom_point() +
         coord_trans(x = "log10", y = "log10")
```



(a) coord_map()

(b) coord_trans(x = "log10", y = "log10")

Figure 2.28 – Quelques systèmes de coordonnées 3

2.1.7 Le facettage ou vignette et groupages

Le facettage ou vignette, permet de de créer de multiples graphiques avec des sous-ensembles du jeu de données créés à partir d'une ou plusieurs variables discrètes. ggplot2, offre 3 fonctions pour cela :

- facet_null() : celui par défaut de tout graphique
- facet_grid() : grille 2D, avec des variables définissant lignes et colonnes,
- facet_wrap() : applique un même graphique à chaque subdivision ou sous-ensemble de données, rangé côte à côte dans une matrice 2x2.

Vignette avec facet_grid()

Elle prend en paramètre le mode de subdivision sous forme de formule ($facteur \sim .$ pour un rangement horizontal,. $\sim facteur$ pour un rangement vertical ou $facteur1 \sim facteur2$ en matrice $facteur1$ en ligne et $facteur2$ en colonne), ensuite des paramètres comme as.table qui gère l'ordre des vignettes, labeller qui permet d'étiqueter chaque vignette (?labellers, pour plus de détails): Par ailleurs, on peut ajouter les situations marginales avec margins = TRUE voir la figure 2.29d. On peut également choisir la variable dont on veut voir les situations marginales en faisant margins = "nom_variable".

```
> p <- ggplot(diamonds)+ geom_point(aes(carat,price))
> p + facet_grid(.~ cut) # rangement vertical
> p + facet_grid(cut ~.)  # rangement horizontal
> p + facet_grid(cut ~ color) #  ligne x colonne
> # ajouter les étiquettes cut et color aux strips
> # afficher les situations marginales : margins = TRUE
> p + facet_grid(cut ~ color, labeller = label_both,
      margins = TRUE)
```

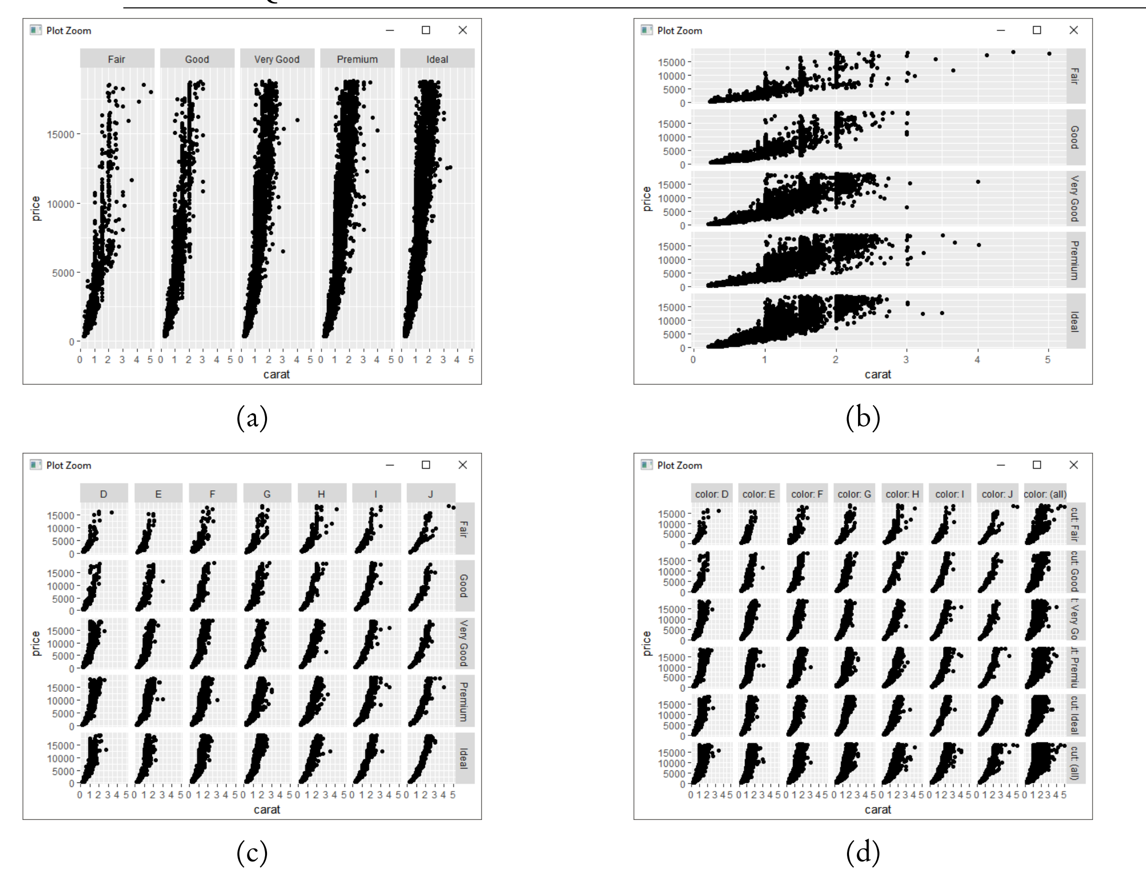


Figure 2.29 – Les vignettes ou facettes avec facet_grid()

Vignette avec facet_wrap()

Cette dernière prend également une formule avec une syntaxe de type $\sim facteur1 + facteur2 + \dots$ ou de type c(*facteur1*,*facteur2*,...) mais fonctionne comme une matrice alors on doit soit spécifier le nombre de colonne **ncol** ou ligne **nrow**.
On a également les paramètres **as.table**, **labeller** et **dir** pour l'ordre de rangement des vignettes.

```
> p + facet_wrap(~ cut, ncol = 2) # facet_wrap
> p + facet_wrap(~ cut, nrow = 2) # ranger par ligne
> p + facet_wrap(~ cut, nrow = 2, dir = "v")
> # pie chart
> ggplot(diamonds) +
        geom_bar(aes(x = cut,fill = cut),width = 1) +
        coord_polar() + facet_wrap(~ color, ncol = 3)
```

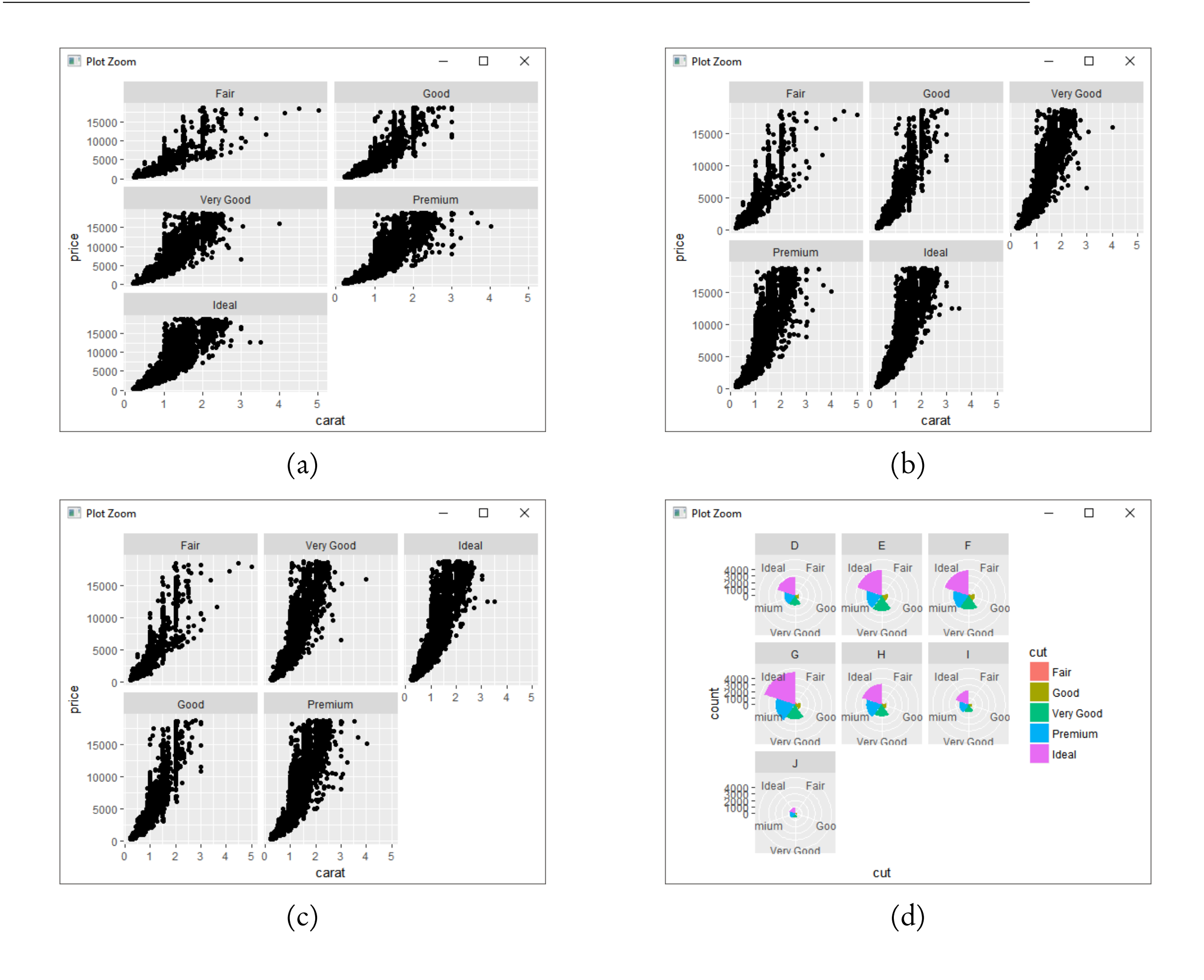


Figure 2.30 – Les vignettes ou facettes avec **facet_wrap()**

Contôler les axes des facettes ou vignettes avec scales

Les échelles des axes peuvent être identiques pour tous les graphiques de la grille ou bien être propres à chaque graphique. C'est le paramètre **scales** qui peut prendre les valeurs suivantes :

- **scales="fixed"** échelles fixes, identiques pour chaque graphique
- **scales="free"** échelles libres, pouvant varier en fonction de chaque graphique
- **scales="free_x"** seule l'échele pour les x peut varier, l'échelle pour les y est fixe
- **scales="free_y"** seule l'échelle pour les y peut varier, l'échelle pour les x est fixe

```
> df <- subset(diamonds,
+        cut == c("Fair","Good","Very Good") &
+        color == c("D" ,"E" ,"F"))
> p <- ggplot(df)+ geom_point(aes(carat,price))
> p + facet_wrap(~cut, scales = "fixed")
> p + facet_wrap(~cut , scales = "free_y")
> p + facet_wrap(~cut , scales = "free_x")
> p + facet_wrap(~cut , scales = "free")
```

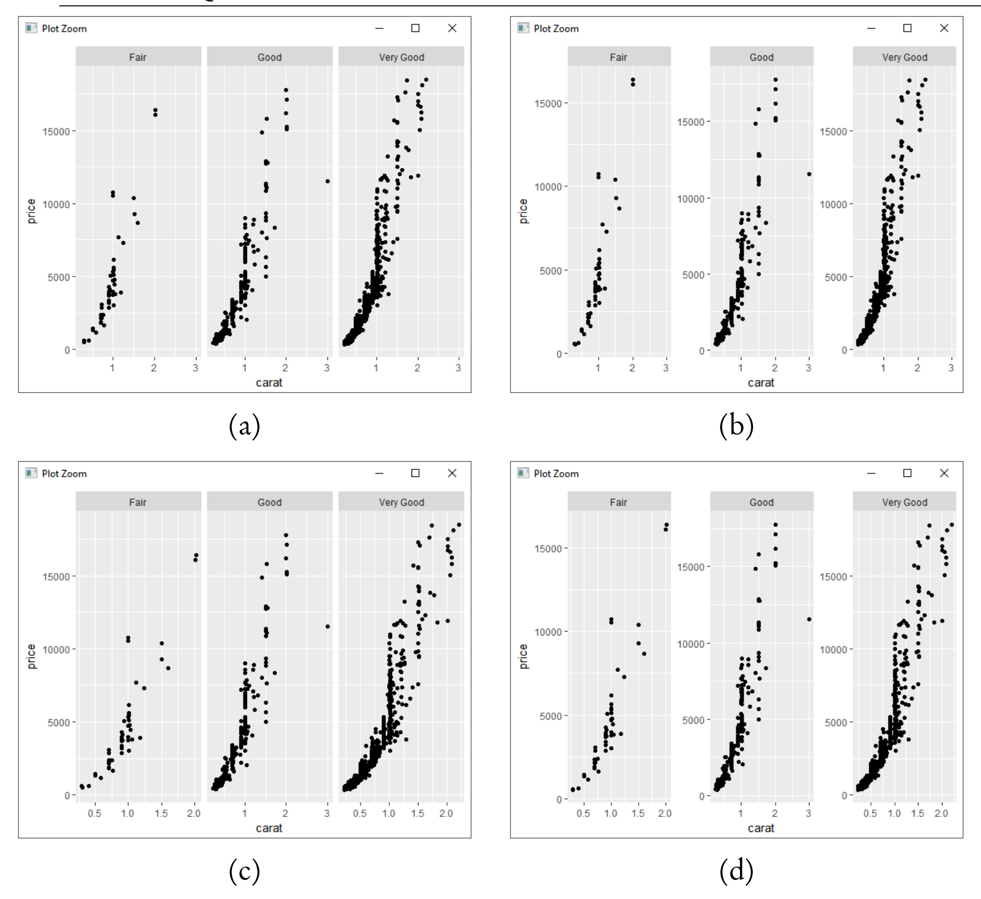


Figure 2.31 – Contrôle des axes des vignettes ou facettes

Des fonctions ggplot2 pour générer des variables catégorielles

Comme les facettes évoque l'idée de sous - ensembles ou division , l'on peut être amener à vouloir diviser ou transformer des variables continues en variables discretes pour réaliser des vignettes ou facettes. ggplot2 offre 3 fonctions pour cela :

- cut_interval(x, n) : découpe la variable continue x en n classe d'effectif égal.
- cut_width(x, width) : découpe la variable continue x en classes d'amplitude width
- cut_number(x, n) : découpe la variable continue x en n classes.

```
> df <- subset(diamonds,
        cut == c("Fair","Good","Very Good") &
        color == c("D" ,"E" ,"F"))
> df$depth_w <- cut_width(df$depth, 5)
> df$depth_i <- cut_interval(df$depth, 5)
> df$depth_n <- cut_number(df$depth, 5)
> p<- ggplot(df, aes(carat,price,colour = price)) +
+       geom_jitter() +
+       scale_color_gradient(low = "blue",high = "red")
> p # défaut
> p + facet_wrap(~ depth_w)
> p + facet_wrap(~ depth_i)
```

```
> p + facet_wrap(~ depth_n)
```

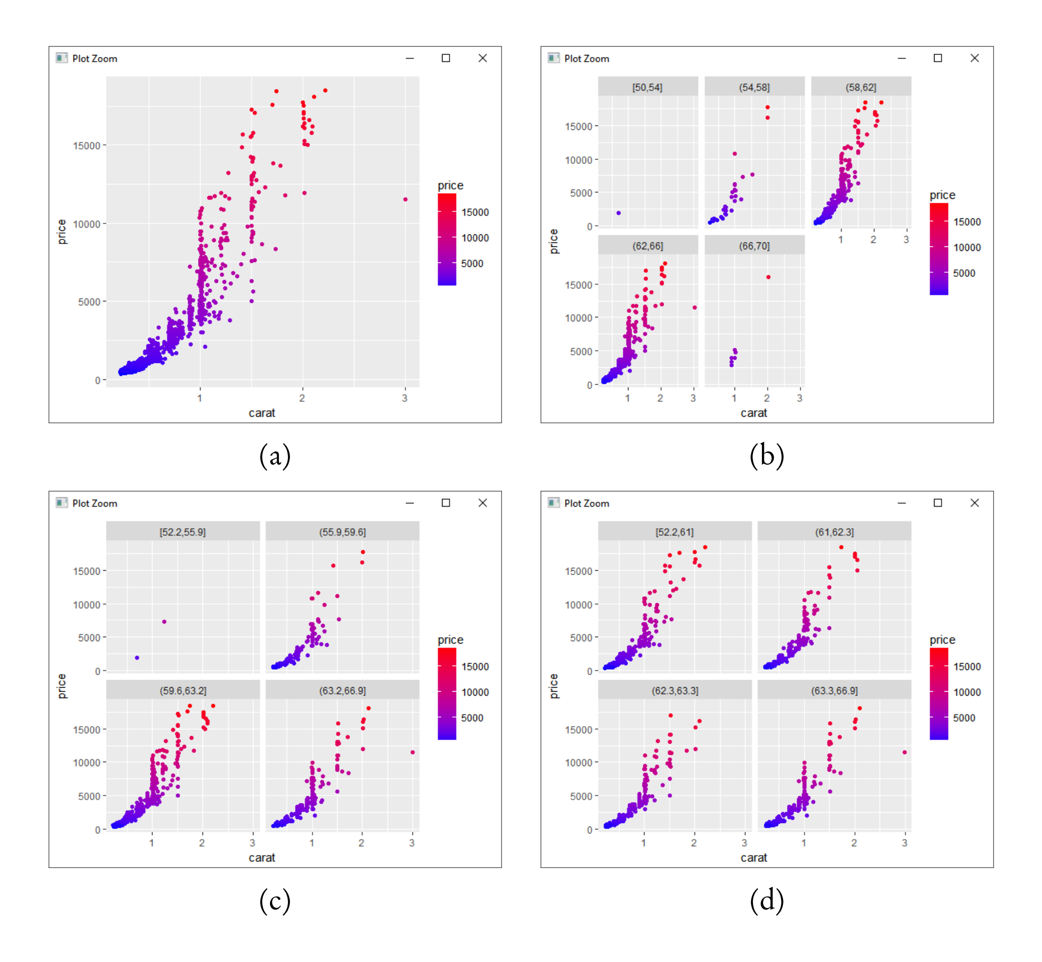


Figure 2.32 – Illustration des fonctions de discrétisation avec facet_wrap()

2.2 Annotation des graphiques

Dans les sections précédentes notamment lorsque l'on a abordé les fonctions d'échelles de position, nous avons montré que ces dernières disposaient du paramètre **name** et **labels** pour respectivement nommer les axes et étiqueter les axes. Mais l'on a également cité les fonctions xlab(), ylab(), labs() et ggtitle() qui sont des alternatives intéressantes. Toutes fois pour des annotations ou ajout de texte autres que les principaux titres. . . , nous avons les fonctions :

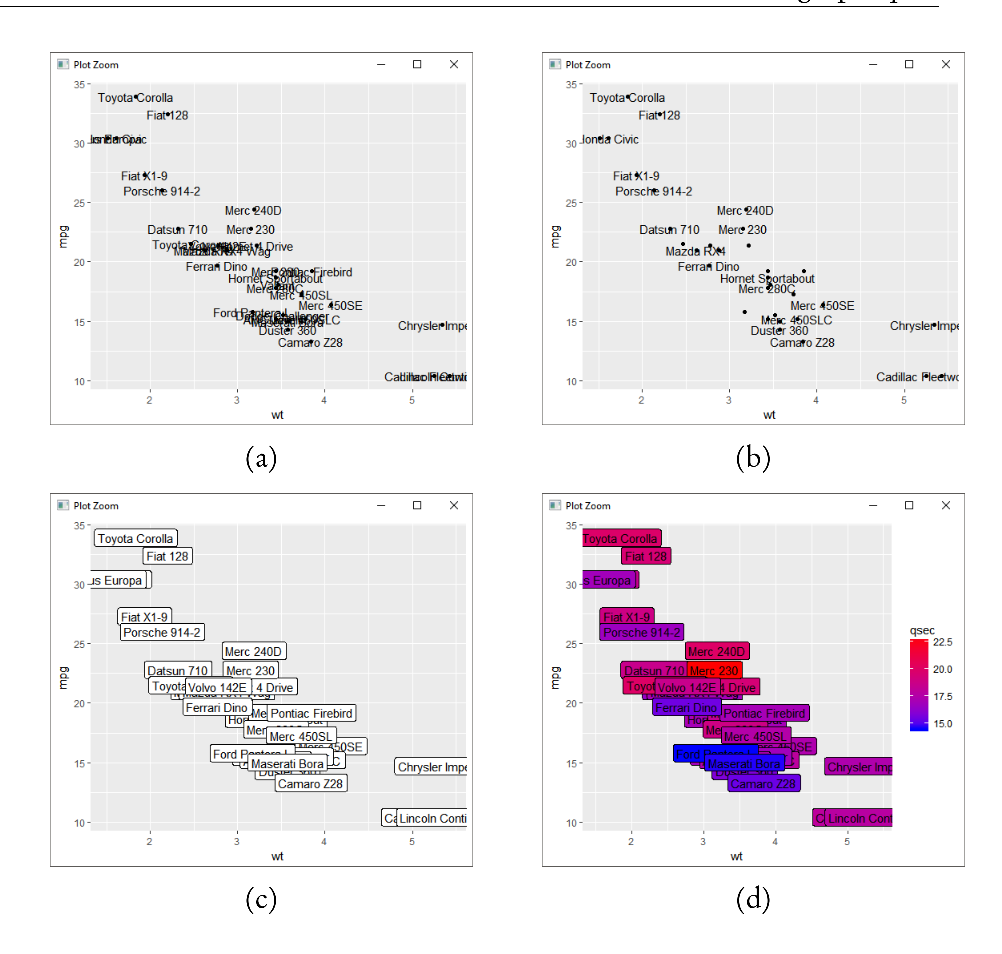
- geom_text() et geom_label(): Pour ajouter du texte à un endroit précis du graphique.
- annotate(): Utile pour ajouter des d'objets variées (texte, flèches . . .)pour annotations à des endroits précis sur le graphique

Elles ont en commun, d'une part les paramètres obligatoires suivants **x,y** et **label** respectivement les coordonnées et le texte à ajouter et d'autre part, les paramètres facultatifs comme angle, colour, family, fontface, group, hjust, lineheight, size, vjust. Particulièrement **hjust** et **vjust**, permettent d'ajuster la position du texte horizontalement (h) et verticalement (v). Si l'on veut passer des formules mathématique en argument, il faut ajouter **parse**= TRUE.

Ajouter du texte avec les fonctions geom_text() et geom_label()

Les deux fonctions permettent d'ajouter du texte notamment relatifs au jeu de données , elles sont capables de gérer les paramètres aesthétiques si ces derniers sont spécifiés sinon elles en héritent de l'objet **ggplot**.

```
> data("mtcars")
> str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76  ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
> df <- cbind(mtcars,lab = rownames(mtcars))
> p <- ggplot(df) + geom_point(aes(wt,mpg))
> p +  geom_text(aes(wt,mpg,label=lab))
> # éviter que les textes soient superposés
> p +  geom_text(aes(wt,mpg,label=lab),check_overlap = TRUE)
> p +  geom_label(aes(wt,mpg,label=lab))
> # remplissage avec une fonction d'échelle
> p +  geom_label(aes(wt,mpg,label=lab,fill=qsec)) +
     scale_fill_gradient(low = "blue",high = "red")
```



(a) (b) (c) (d)

Figure 2.33 – Annotation avec **geom_text()** et **geom_label()**

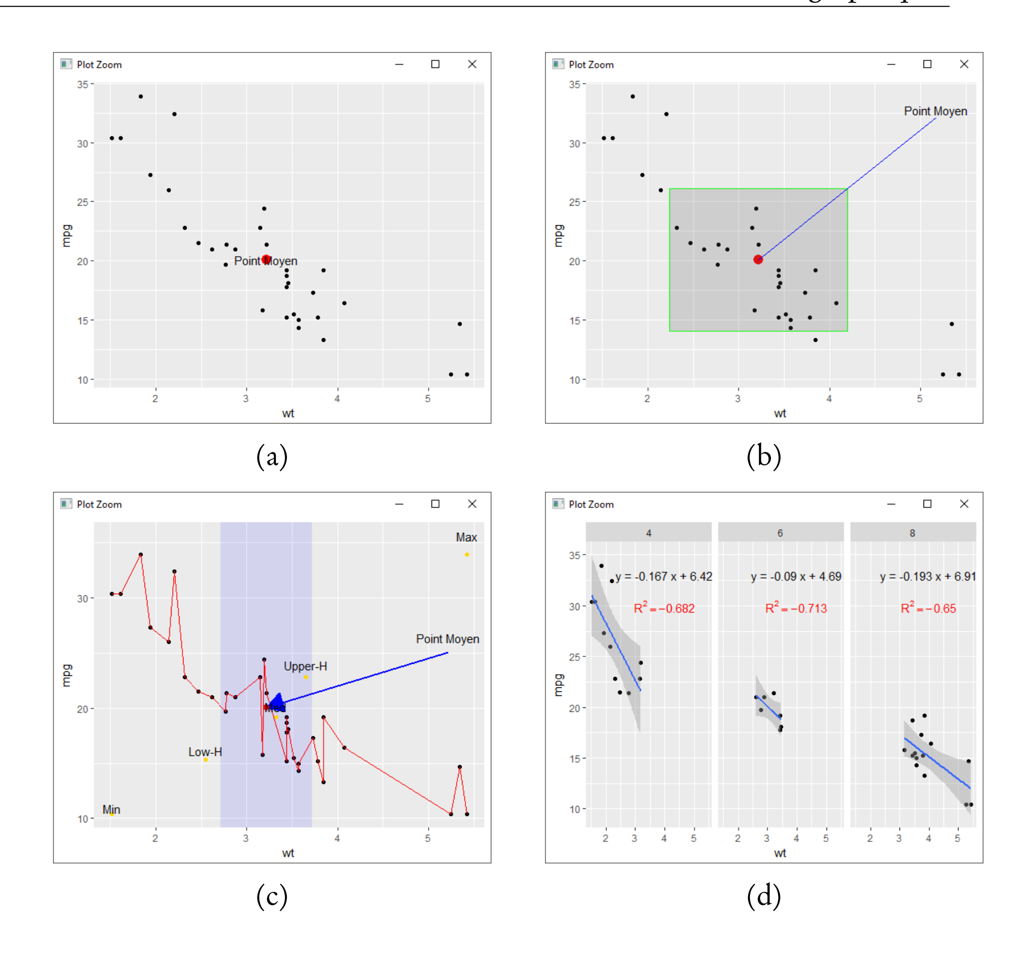
La fonction **geom_label()** est plus lente que **geom_text()** mais en contrepartie, offre la possibilité de mettre du texte encadré avec en sus le paramètre **fill** pour agir sur la couleur de remplissage comme on peut le voir sur la figure 2.33d.

Annotation avec la fonction annotate()

annotate() sert à non seulement ajouter du texte mais également d'autres objets comme , les flèches, les segments. . . C'est le paramètre **geom**, qui permet de spécifier le type d'objet, outres les autres paramètres communs aux fonctions d'annotations citer en début de section, nous devons fournir les paramètres spécifiques à l'objet choisi c'est - à - dire que si l'on choisi un rectangle il faudrait **xmin**, **xmax**, **ymin**, **ymax**.

```
> p + geom_point(x = mean(df$wt), y = mean(df$mpg),
    size=4,colour = "red") +
    annotate(geom = "text",x = mean(df$wt),y = mean(df$mpg),
            label = "Point Moyen")
> p + geom_point(x = mean(df$wt), y = mean(df$mpg),
                size=4, colour = "red") +
    # ajouter un rectangle
    annotate(geom = "rect", xmin = mean(df$wt)- sd(df$wt),
    xmax = mean(df$wt) + sd(df$wt), ymin = mean(df$mpg) - sd(
      df$mpg), ymax = mean(df$mpg) + sd(df$mpg),
            alpha = .2, colour = "green") +
```

```
  # ajouter un segment reliant le point moyen et le texte
  annotate(geom = "segment",x = mean(df$wt),y = mean(df$mpg)
     ,
           xend = mean(df$wt) +  2*sd(df$wt),
     yend = mean(df$mpg) + 2*sd(df$mpg),colour = "blue") +
  annotate(geom = "text", x = mean(df$wt)+ 2*sd(df$wt),
  y = mean(df$mpg)+2.1*sd(df$mpg),label = "Point Moyen")
> # ajouter une bande
> p + annotate("rect",xmin = mean(df$wt) - 0.5,
           xmax = mean(df$wt) + 0.5, alpha = 0.1,
           ymin = -Inf, ymax = Inf, fill = "blue") +
    # ajouter une courbe des points
   geom_line(aes(wt,mpg), colour="red",size = 0.7) +
   # ajouter un segment et lui attribuer une flèche
   annotate(geom = "segment",x =mean(df$wt)+2,
           y=mean(df$mpg) + 5, xend = mean(df$wt),
           yend = mean(df$mpg),colour = "blue",
           arrow = arrow(type ="closed"),size = 0.8) +
    # ajouter les 5 paramètres d'un box-and-whisker & la
      moyenne
   annotate("point", x = c(fivenum(df$wt),mean(df$wt)),
                    y = c(fivenum(df$mpg),mean(df$mpg)),
                    colour = c(rep("gold",5),"red"),
                    size = c(rep(1.5,5),2.5)) +
   annotate("text", x = c(fivenum(df$wt),mean(df$wt)+2),
                   y = c(fivenum(df$mpg),mean(df$mpg)+5)*
                      1.05,
            label= c("Min", "Low-H","Med","Upper-H","Max",
              "Point Moyen"))
> # une routine pour générer des équations lm en texte
> # et les coefficients de correlation
> reg <- NULL
> rcoef <- NULL
> for( l in unique(df$cyl)){
   data <- df[df$cyl == l,]
   md <- lm(wt ~ mpg,data)
   reg<- c(reg,paste("y =",round(md$coefficients[2],3),"x",
   ifelse(md$coefficients[1]<0,'',"+"),
     round(md$coefficients[1],2),sep=' '))
     rcoef<- c(rcoef,paste("R^2==",
     round(cor(data$wt,data$mpg),3),sep=' '))}
> p + geom_smooth(aes(wt,mpg),method = "lm") +
   facet_grid(.~cyl) + annotate("text",x = 4,y=33,
   label= reg) + annotate("text",x=  4,y =30,label= rcoef,
   colour ="red", parse = TRUE)
```



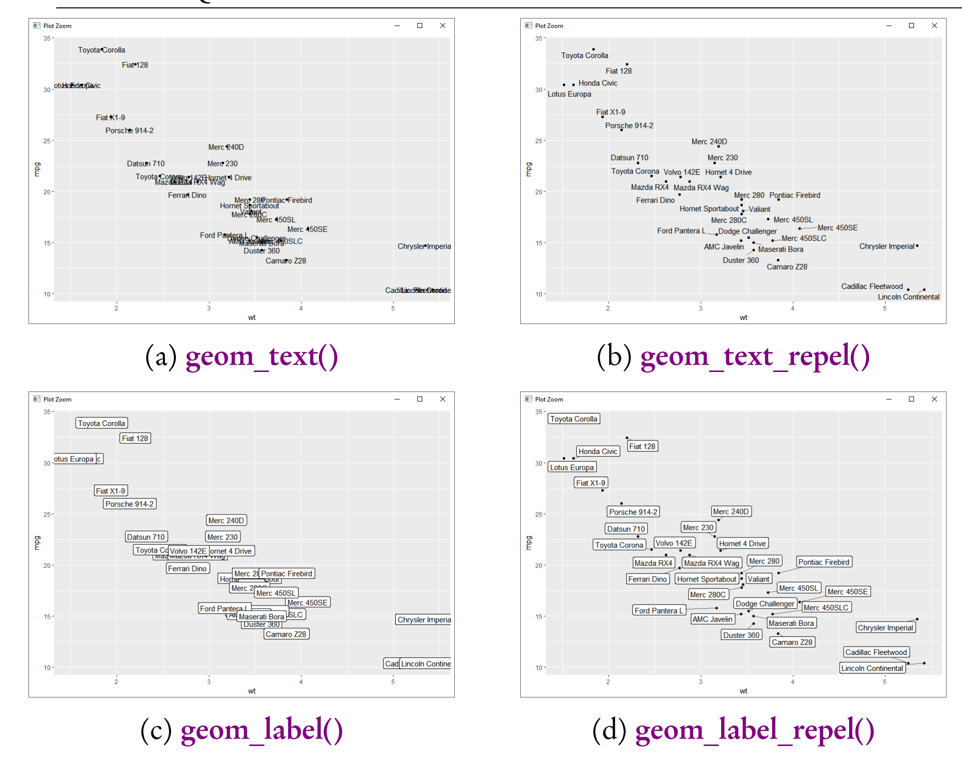
(a) (b) (c) (d)

Figure 2.34 – Annotation avec annotate()

Annotation avec le package ggrepel

Ce package a été conçu spécialement pour que le labelling de texte se réalise efficacement. En effet, Il offre les fonctions geom_text_repel() et geom_label_repel() alternatives à geom_label() et geom_text(), qui permettent d'éviter de façon optimale le chevauchement des textes ajoutés.

```
> library(ggrepel)
> p + geom_text(aes(wt, mpg, label = lab))
> p + geom_text_repel(aes(wt, mpg, label = lab))
> p + geom_label(aes(wt, mpg, label = lab))
> p + geom_label_repel(aes(wt, mpg, label = lab))
```



(a) geom_text() (b) geom_text_repel()

(c) geom_label() (d) geom_label_repel()

Figure 2.35 – Annotation avec geom_text_repel() et geom_label_repel()

A ce niveau il se peut que l'on ait un problème de compatibilité et obtienne l'erreur suivante:

```
> p + geom_text_repel(aes(wt, mpg, label = lab))
Error: GeomTextRepel was built with an incompatible version
   of ggproto.
```

Pour la résoudre, une manière est de réinstaller le package, mais avant, il faut supprimer le package :

```
remove.packages("ggrepel")
install.packages("ggrepel", type = "source")
```

Pour des annotations plus avancées notamment des annotations spécifiques à chaque vignette ou facette ou ajout d'image en arrière plan voir 3.10.

2.3 Thèmes et Légendes

Thèmes

L'on sait maintenant presque tout personnaliser avec tous ces aspects de **ggplot2**, qu'on a abordé dans les sections précédentes. Toutefois, nous n'avons pas encore appris comment changer l'apparence globale du graphique. En effet, ce rôle est réservé aux thèmes. Jusqu'ici, nous avons utilisé le thème par défaut **theme_grey()** (d'où le fond gris). Il n'est pas terrible mais la plupart du temps, il suffit. Sinon, dans cette section nous allons découvrir d'autres thèmes et également apprendre à les personnaliser pour notre utilisation ou carrément apprendre à concevoir nos propres thèmes.

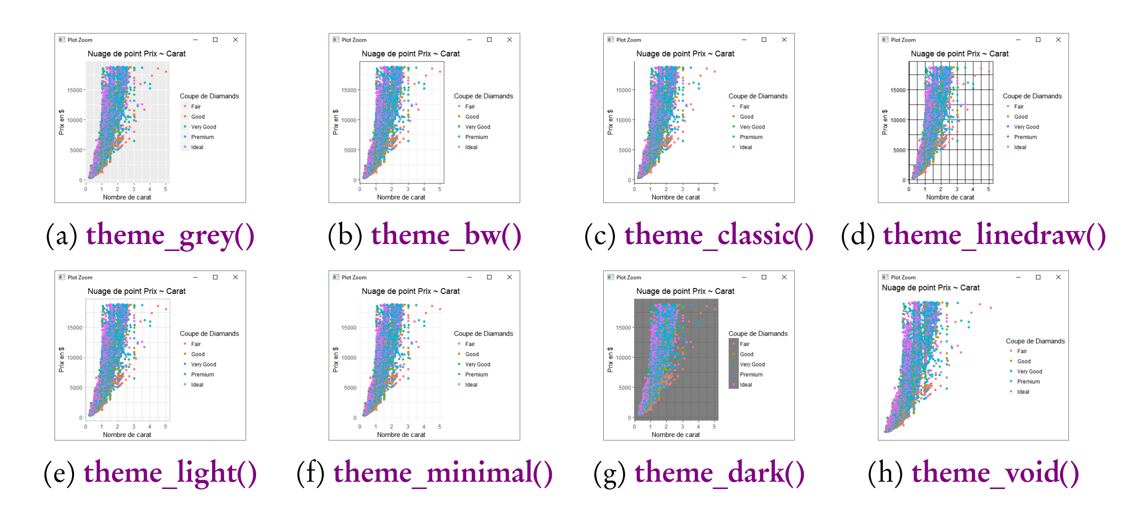
Découverte et utilisation des thèmes

L'autre thème disponible avec **ggplot2** c'est le thème **theme_bw()** (pour Black-and-White), **theme_classic()** . . . (au total 8 comme les présente la figure 2.36). Comme les autres fonctions les thèmes s'ajoutent à l'objet **ggplot**, selon le modèle suivant :

```
> # impacte le graphique actuel
> p + theme_xxx()
> # impacte les graphiques qui viennent après
> # au cours de  la session
> theme_set(theme_xxx())
```

Si nous ne voulons pas à chaque graphique ajouter un thème, nous pouvons définir un thème par défaut avec la fonction **theme_set()**. Découvrons quelques thèmes :

```
> p <- ggplot(diamonds) + geom_point(aes(x = carat,y = price)
  )
> # thème par défaut
> p + theme_grey()
> # fond blanc + grille
> p + theme_bw()
> # sans grille d'arrière plan
> p + theme_classic()
> # seule la grille d'arrière plan
> p + theme_linedraw()
> # fond clair et grille grise
> p + theme_light()
> # thème minimal sans couleur d'arrière plan
> p + theme_minimal()
> # arrère plan sombre
> p + theme_dark()
> # un thème vide ! seulement les objets géométriques
> p + theme_void()
```



(a) **theme_grey()** (b) **theme_bw()** (c) **theme_classic()** (d) **theme_linedraw()**

(e) **theme_light()** (f) **theme_minimal()** (g) **theme_dark()** (h) **theme_void()**

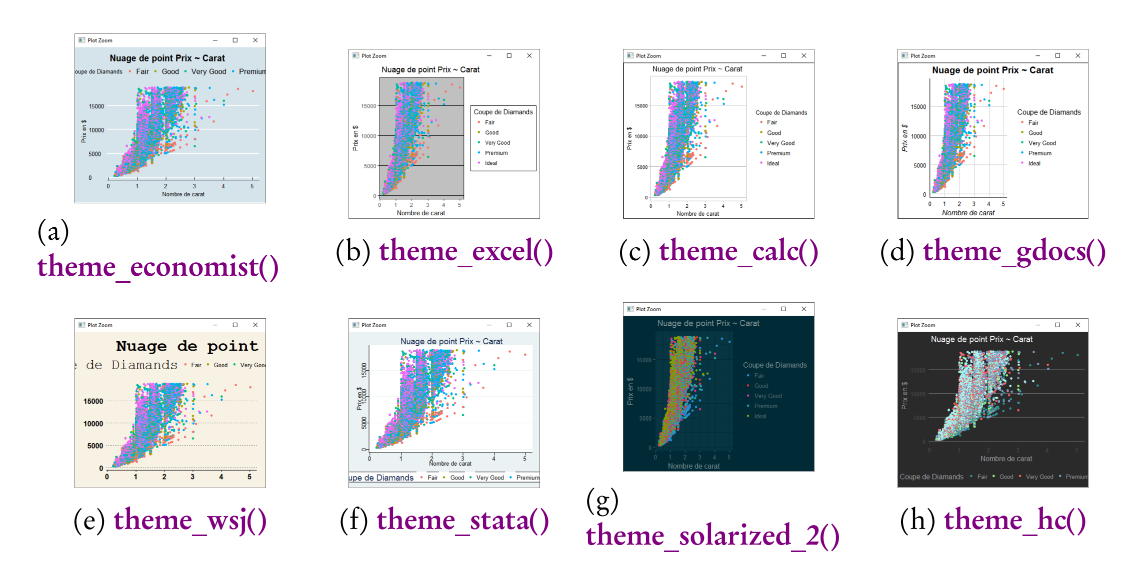
Figure 2.36 – Les thèmes disponible dans **ggplot2**

Et si ces thèmes, ne vous conviennent pas nous avons le package **ggthemes** de **Jeffrey**

Arnold (Page Github de ggthemes). Ce package contient des thèmes d'une élégance à vous mettre dans l'embarras (voir 2.37) :

```
> library(ggthemes)
> # thème graphique du Magazine L'Economist
> p + theme_economist()
> # thème graphique de MSOffice Excel
> p + theme_excel()
> # thème graphique de LibreOffice Calc
> p + theme_calc()
> # thème graphique de google docs
> p + theme_gdocs()
> # thème graphique de The Wall Street Journal.
> p + theme_wsj()
> # thème graphique du logiciel Stata
> p + theme_stata()
> # thème graphique basé sur la palette solorized
> p + theme_solarized_2(light = FALSE) +
  scale_colour_solarized("blue")
> # thème graphique basé sur http://www.highcharts.com/
> p + theme_hc(bgcolor = "darkunica") +
  scale_colour_hc("darkunica")
```

Par ailleurs comme on peut le voir dans le code ci-dessus, le package ne contient pas que des thèmes mais également des fonctions scales ou d'échelles pour raffiner les couleurs, le remplissages,. . .



(a) theme_economist() (b) theme_excel() (c) theme_calc() (d) theme_gdocs()

(e) theme_wsj() (f) theme_stata() (g) theme_solarized_2() (h) theme_hc()

Figure 2.37 – Les thèmes disponible dans ggplot2

Les éléments d'un thème

Il existe 40 éléments qui permettent d'avoir un contrôle total sur l'apparence d'un graphique à travers un thème. Ces éléments peuvent être regroupés selon la partie du graphique qu'ils impactent :

1. Les éléments de la zone de graphique(plot):

plot.element	element_function	Description
plot.background	element_rect()	arrière plan graphique
plot.title	element_text()	titre du graphique
plot.margin	margin()	marge au tour du graphique

2. Les éléments des axes :

axis.element	element_function	Description
axis.line	element_line()	axes
axis.text	element_text()	étiquette de graduation
axis.text.x	element_text()	étiquette de graduation axe des x
axis.text.y	element_text()	étiquette de graduation axe des y
axis.title	element_text()	les titres des axes
axis.title.x	element_text()	les titres des axes des x
axis.title.y	element_text()	les titres des axes des y
axis.ticks	element_line()	les marques de gradution des axes
axis.ticks.length	unit()	Longueur des marques de gradution des axes

3. Les éléments de la légende :

legend.element	element_function	Description
legend.background	element_rect()	arrière plan de la légende
legend.key	element_rect()	arrière plan des symboles de la légende
legend.key.size	unit()	la taille des symboles de la légende
legend.key.height	unit()	la hauteur des symboles de la légende
legend.key.width	unit()	la largeur des symboles de la légende
legend.margin	unit()	la marge de la légende
legend.text	element_text()	les étiquettes de la légende
legend.text.align	0–1	l'allignement de des étiquettes de la légende(0 pour droit, 1 pour gauche)
legend.title	element_text()	titre de la légende
legend.title.align	0–1	l'allignement du titre de la légende(0 pour droit, 1 pour gauche)

4. Les éléments de la zone de tracé

panel.element	element_function	Description
panel.background	**element_rect()**	arrière plan de zone de tracé
panel.border	**element_rect()**	bordure de la zone de tracé
panel.grid.major	**element_line()**	les lignes majeures de la grille de la zone de tracé
panel.grid.major.x	**element_line()**	les lignes verticales majeures de la grille de la zone de tracé
panel.grid.major.y	**element_line()**	les lignes horizontales majeures de la grille de la zone de tracé
panel.grid.minor	**element_line()**	les lignes mineures de la grille de la zone de tracé
panel.grid.minor.x	**element_line()**	les lignes verticales mineures de la grille de la zone de tracé
panel.grid.minor.y	**element_line()**	les lignes horizontales mineures de la grille de la zone de tracé
aspect.ratio	**numeric**	le ratio d'aspect

5. Les éléments de vignettes ou facettes :

strip.element	element_function	Description
strip.background	**element_rect()**	arrière plan des entêtes de vignettes
strip.text	**element_text()**	textes des entêtes de vignettes
strip.text.x	**element_text()**	textes des entêtes de vignettes horizontalement rangées
strip.text.y	**element_text()**	textes des entêtes des vignettes verticalement rangées
panel.spacing	**unit()**	marge entre vignettes
panel.spacing.x	**unit()**	marge entre vignettes verticalement rangées
panel.spacing.y	**unit()**	marges entre vignettes horizontalement rangées

Modification et conception de thème

Avant d'arriver à la modification et la conception de thème, il est important de maîtriser la structure ou plus précisément les différentes catégories de paramètres qui constituent un thème, énumérées précédemment. C'est à travers la fonction **theme()**, qu'on modifie en paramètre les éléments ou caractéristiques d'un thème selon le modèle suivant :

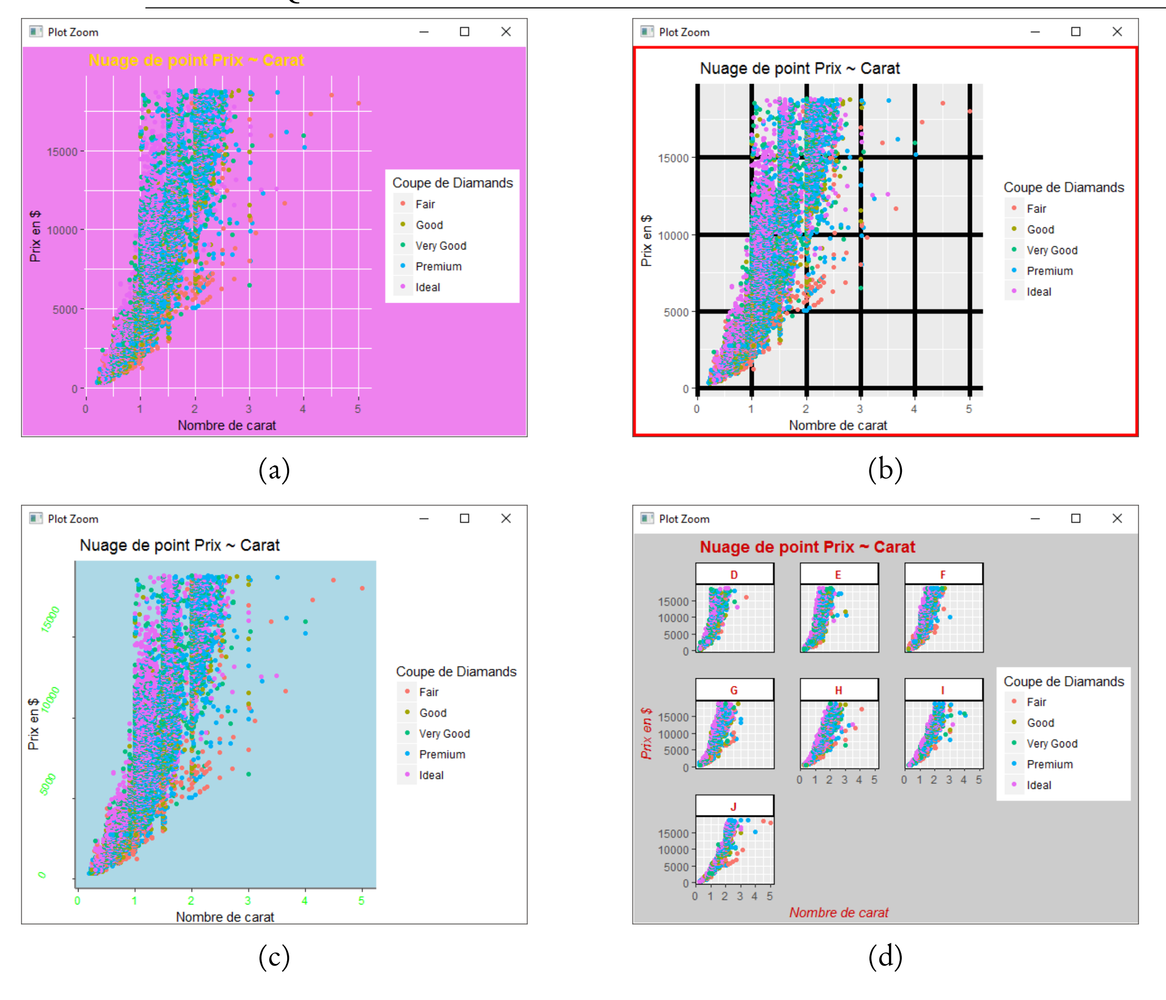
```
# modification du thème pour le graphique actuel
p + theme(element.nom = element_function())
# modification du thème pour le reste des
# graphiques de la session
theme.modifie <-
theme_update(element1.nom = element_function(),
             element2.nom = element_function(),
                    ...
             elementn.nom = element_function())
theme_set(theme.modifie)
```

La fonction **theme_update()**, permet de modifier ou mettre à jour le thème actuel pour que ce dernier prenne en compte les modifications et on peut utiliser **theme_set()**, pour définir un thème pour le reste de la session. Nous avons listé précédemment les fonctions relatives à chaque éléments de thème et ces fonctions se résument à ces 4 fonctions :

- **element_text**(**family**, **face**, **colour**, **size**, **hjust**, **vjust**, **angle**,**margin** ...) pour modifier tous les éléments de type texte comme les titres, les étiquettes ou labels. Particulièrement le paramètre margin prend une fonction **margin**(**t**=, **l**=, **b**=, **r**=) qui permet de définir une marge au tour du texte (titre, labels) où **t** signifie **top**, **l** pour **left** ...
- **element_line**(**colour**, **size**, **linetype**) permet de définir les lignes notamment celles de la grille et des axes.
- **element_rect**(**fill**, **colour**, **size**, **linetype**...) permet de définir les rectangles ou cadre notamment pour encadrer l'arrière plan, zone de tracé...
- **element_blank()**, cette dernière permet de ne rien définir !

Ci - dessous, nous présentons quelques utilisations de ces fonctions pour modifier les thèmes. Nous n'avons pas abordé le cas des éléments de la légende par ce que nous l'abordons dans la sous section suivante.

```
# changer l'arrière-plan et le titre
> p + theme(plot.title = element_text(face = "bold",
                  colour = "gold"),
  plot.background = element_rect(fill = "violet"),
  panel.background = element_blank())
# modifier des éléments de la zone graphique et de tracé
> p + theme(
  plot.title = element_text(margin = margin(t = 7,b = 7)),
  plot.background = element_rect(colour = "red", size = 2),
  panel.grid.major = element_line(colour = "black",size = 2))
# changer la marge et les éléments des axes
> p + theme(axis.line=element_line(colour = "grey50",size =
   1),
  axis.text.y=element_text(angle = 60, hjust = -0.1,
                                   vjust = 0.07),
  panel.background = element_rect(fill = "lightblue"),
  plot.margin = margin(4, 2, 2, 4),
  panel.grid = element_blank(),
  axis.text = element_text(color = "green"))
> # modification des éléments vignettes
> p + facet_wrap(~ color) +
  theme(panel.spacing = unit(0.3, "in"),
  strip.background = element_rect(fill = "white",
                       color = "black", size = 1),
  strip.text = element_text(colour = "red3",face = "bold"),
  panel.border = element_rect(fill = NA),
  plot.background = element_rect("gray80"),
  plot.title = element_text(colour = "red3",face = "bold"),
  axis.title = element_text(colour = "red3",face = "italic"))
```



(a) (b)

(c) (d)

Figure 2.38 – Modification des thèmes

Pour concevoir son propre thème, il suffit d'utiliser un thème de base et le personnaliser. Comme ceci :

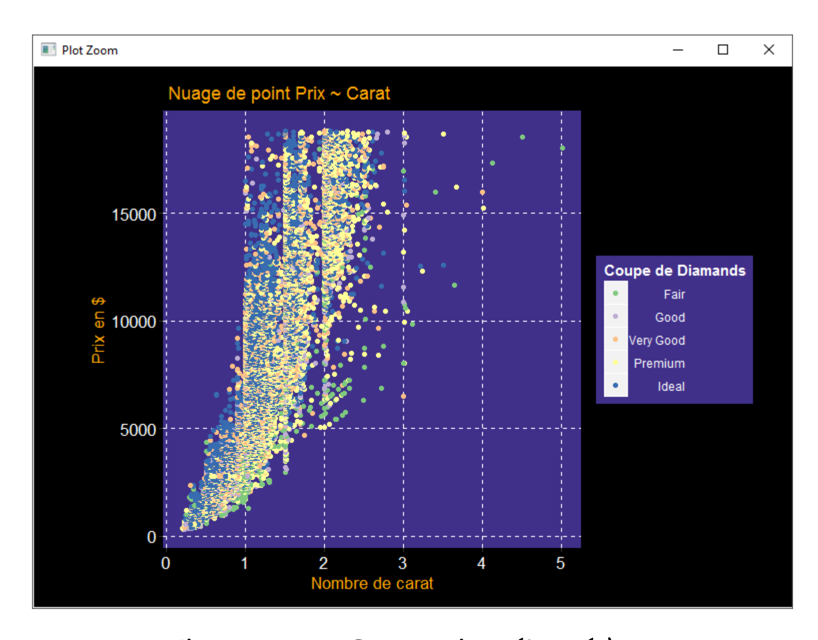


Figure 2.39 – Conception d'un thème

```
> mon.theme <- theme(plot.background=element_rect(fill="black
  "),
     plot.margin = unit(c(1, 2, 1, 3), "lines"),
     plot.title = element_text(colour="orange",
                               margin = margin(b = 7)),
     panel.background = element_rect(fill="#40308A"),
     panel.grid.minor = element_blank(),
     panel.grid.major = element_line(linetype=2,
                                    colour = "white"),
     axis.text = element_text(size=13, colour="white"),
     axis.title.x = element_text(size=13,colour="orange"),
     axis.title.y = element_text(size=13,colour="orange"),
     legend.background = element_rect(fill="#40308A"),
     legend.text = element_text(colour = "white" ),
     legend.title = element_text(colour = "white",
                                 face = "bold"),
     legend.text.align = 1)
> p + scale_color_brewer(palette = "Accent") + mon.theme
```

Si l'on tape **theme_gray**, sans les parenthèses, on peut voir l'ensemble des éléments et leur valeur par défaut, cette fonction peut ensuite servir comme base de conception de nos propres thèmes.

Légendes

En effet, selon l'élément à modifier sur un thème correspond une fonction d'élément. On distingue notamment des éléments textes, géométriques . . . Les légendes sont automatiquement générées par **ggplot2** et pour la plus part du temps cela s'avère suffisant. En effet, si l'on a prêté attention aux graphiques élaborés jusqu'ici, les légendes dépendent essentiellement de la spécification des aesthetiques(couleur, taille, forme, remplissage).
Le titre et les étiquettes de la légende sont automatiquement générés selon le nom des variables utilisées.

Modification de Titres et étiquettes de légende

Pour définir soi-même le titre et les étiquettes de la légende, l'on dispose des fonctions **labs()**, les fonctions **scale_xxx()**. En effet, comme mentionnée en introduction de cette sous section, les légendes dépendent entièrement des esthétiques. Ainsi, pour définir le titre d'une légende il faut soit renommer l'aesthétique dans la fonction **labs()** avec les paramètres **fill**, **colour**, **shape**. . . ou dans la fonction d'échelle ou scale **scale_fill_xxx()**, **scale_colour_xxx()**, **scale_shape_xxx()** avec l'argument **name**. Par contre pour modifier les étiquettes de la légende il faut utiliser le paramètre **label** dans la fonction scale correspondante scale **scale_fill_xxx()**, **scale_colour_xxx()**, **scale_shape_xxx()**.

```
> df <- subset(diamonds,
      cut == c("Fair","Good","Very Good") &
      color == c("D" ,"E" ,"F"))
> p <- ggplot(df)+ geom_point(aes(carat,price,
                  colour = color, shape = cut))
> p
```

```
> p + labs(shape = "Les formes \n avec Fill",
      colour = "Coloration")
> # ou avec des fonctions scales
> p + scale_shape_discrete(name = "Les formes\n avec Scale")
 +
     scale_color_discrete(name = "Coloration avec sale ")
> p + scale_shape_discrete(name = "Les formes \n avec Scale",
      label = c("Passable","Bon", "Très Bon")) +
     scale_color_discrete(name = "Coloration avec sale ",
      label = c("Couleur D","Couleur E","Couleur D"))
```

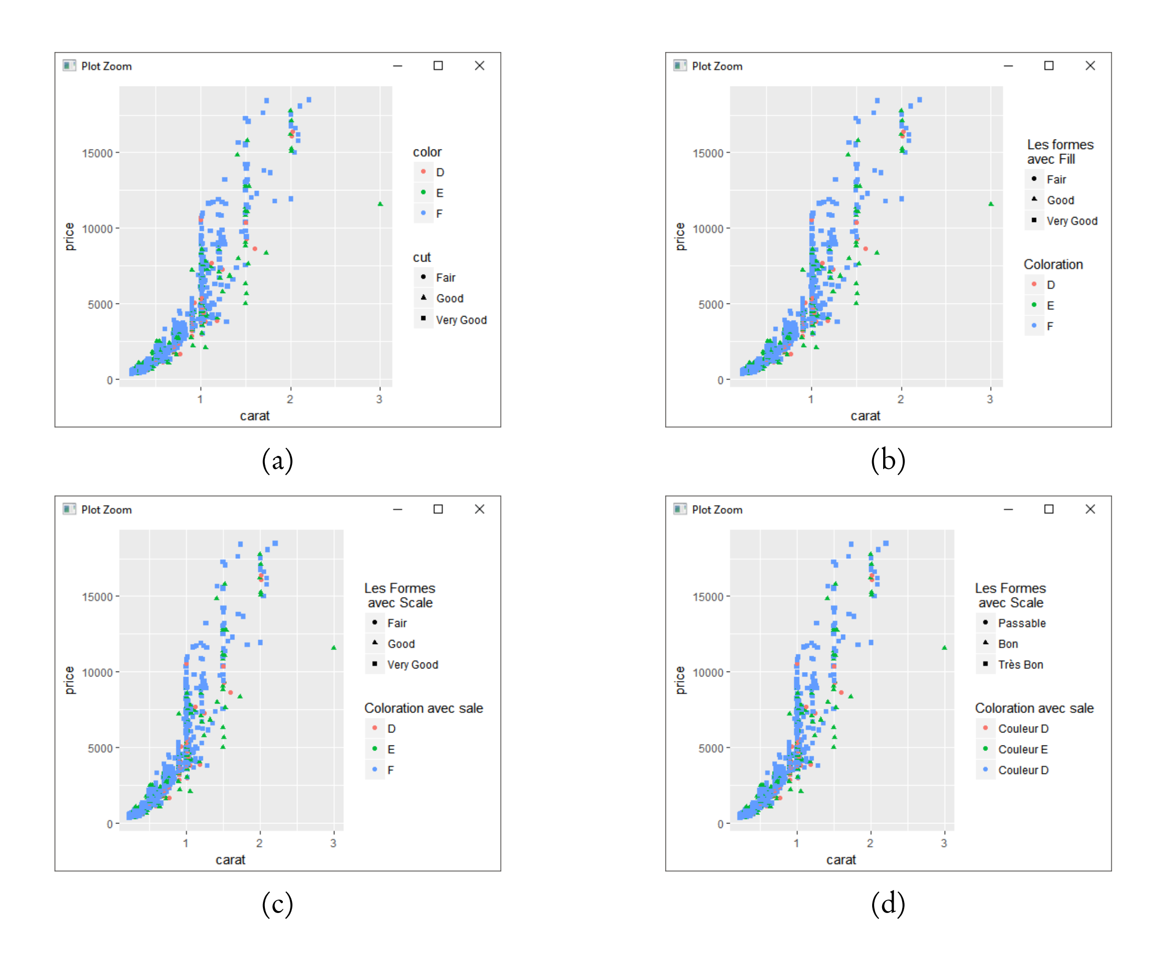


Figure 2.40 – Modification des titres et étiquettes

Positionnement de la légende

Le positionnement d'une légende s'effectue au niveau de la fonction **theme()** avec son paramètre **legend.position** qui peut prendre les valeurs : **"right"**, **"left"**, **"top"**, **"bottom"**, ou **"none"** (ce dernier supprime la légende). Toutefois, pour plus de flexibilité et si l'on veut placer la légende dans la zone de tracé plutôt que dans la marge , ce paramètre peut admettre un vecteur contenant les coordonnées (x,y) sur une échelle unitaire. Ce qui veut dire que **legend.position = c(0,1)** équivaut au coin supérieur gauche.

```
> p <- ggplot(df)+ geom_point(aes(carat,price,
                      colour = color))
> p # par défaut à droite
```

```
> p + theme(legend.position = "top")
> p + theme(legend.position = "left")
> p + theme(legend.position = "bottom")
> # légende au coin supérieur droit avec justification
> # du coin droit supérieur de la légende
> p + theme(legend.position = c(0.9, 0.9),
          legend.justification = c(1,1))
> # légende au coin inférieur droit avec
> # justification coin inférieur droit de la
> # légende
> p + theme(legend.position = c(0.9, 0.1),
          legend.justification = c(1,0))
> # légende au centre avec justification centrée
> p + theme(legend.position = c(0.5, 0.5),
    legend.justification = c(0.5, 0.5))
> # modification de la direction
> p + theme(legend.position = c(0.8, 0.4),
    legend.justification = c(0.5, 0.5),
    legend.direction =  "horizontal")
```

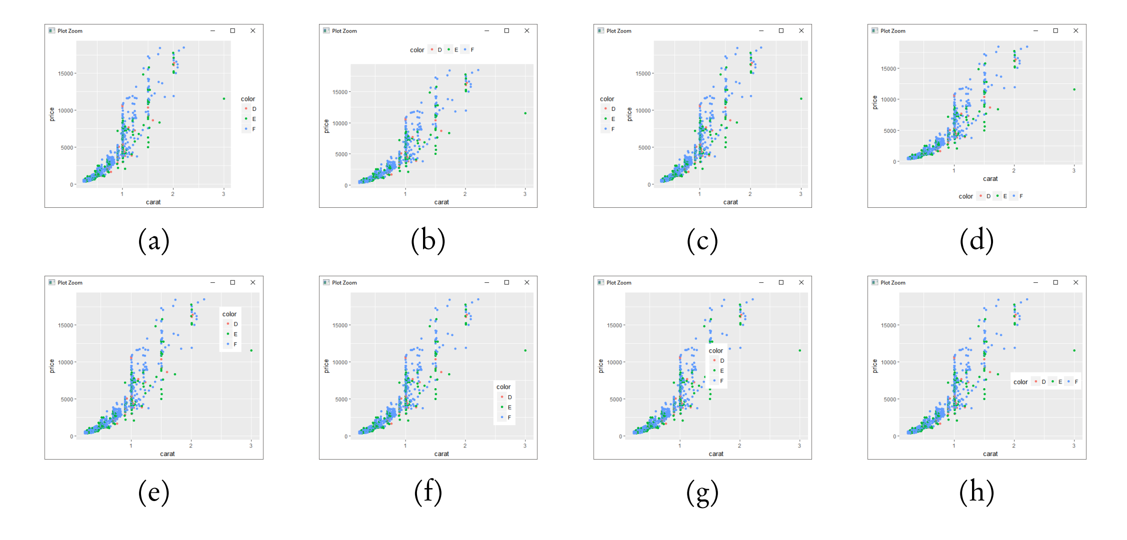


Figure 2.41 – Positionnement de la légende

Par ailleurs, les paramètres comme **legend.justification** (prend un vecteur à l'échelle unitaire) et **legend.direction** permettent respectivement de spécifier quelle partie de la légende doit être placée à l'emplacement spécifié par **legend.position** et de spécifier l'orientation de la légende("horizontal" ou "vertical").

Apparence de la légende

Autres modifications possibles: l'apparence de la légende et les couleurs, polices des titres et étiquettes. Pour se faire, il faut définir les arguments **legend.title** et **legend.text** de la fonction **theme()**, avec la fonction **element_text**(**face**=, **family**=, **colour**=,**size**...). De même **legend.background** et **legend.key** permettent avec la fonction **element_rect**(**fill**=, **colour**=, **size**=) permet de changer l'apparence d'arrière plan de la légende.

```
> p + theme(legend.key = element_rect(fill ="yellow",
                             colour = "black"),
       legend.background = element_rect(fill = "seagreen"))
> p + theme(legend.title = element_text(face="bold",
                                    size = 12,
                                    colour = "red"),
          legend.text = element_text(face = "italic",
                                    colour ="blue"))
> p + guides(colour = guide_legend(ncol = 2),
          shape = guide_legend(ncol = 2, byrow = TRUE))
> # on peut même definir les étiquettes avec guides()
> p + guides(colour = guide_legend(reverse = TRUE,
                 title = "Coloration"),
          shape = guide_legend(ncol = 2, title = "forme"))
```

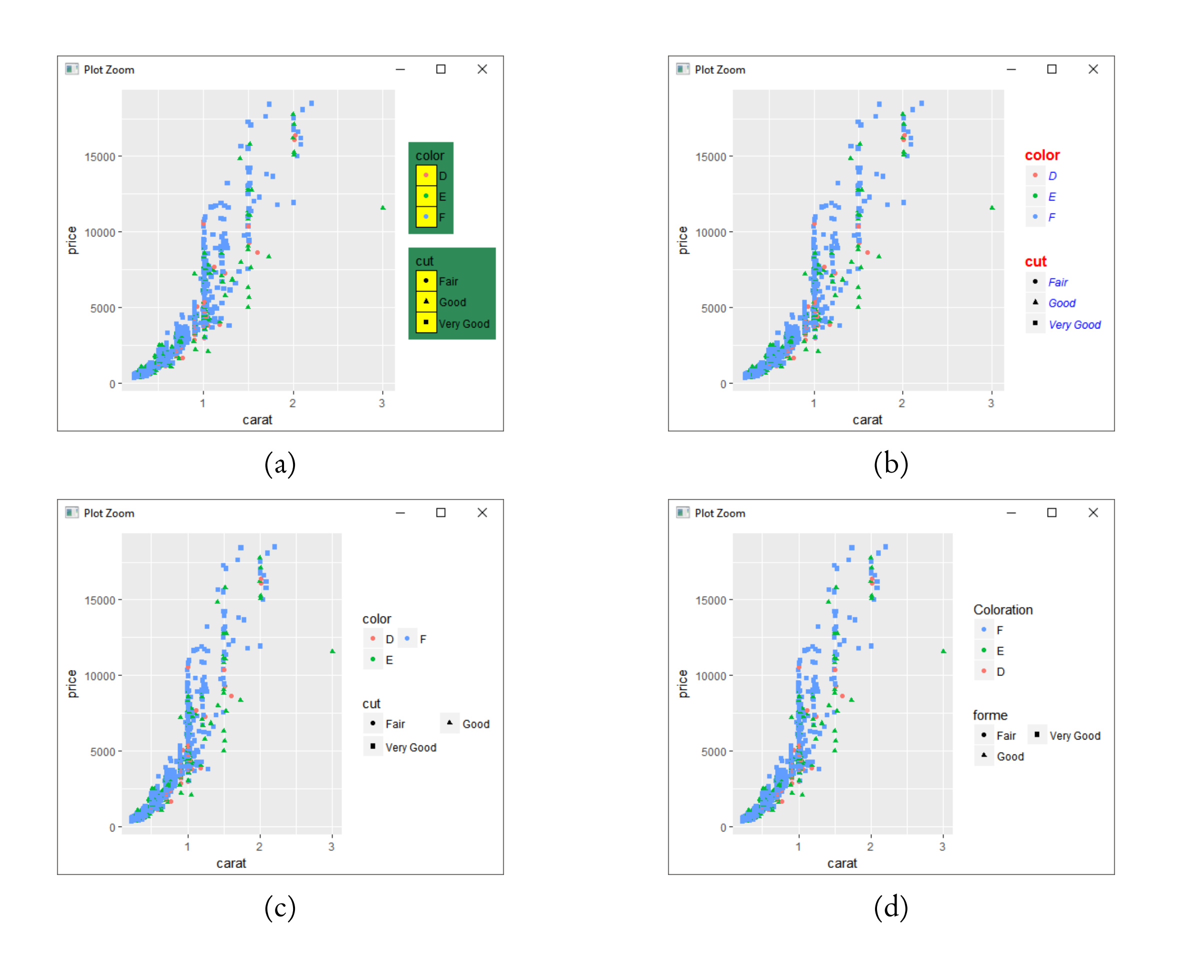


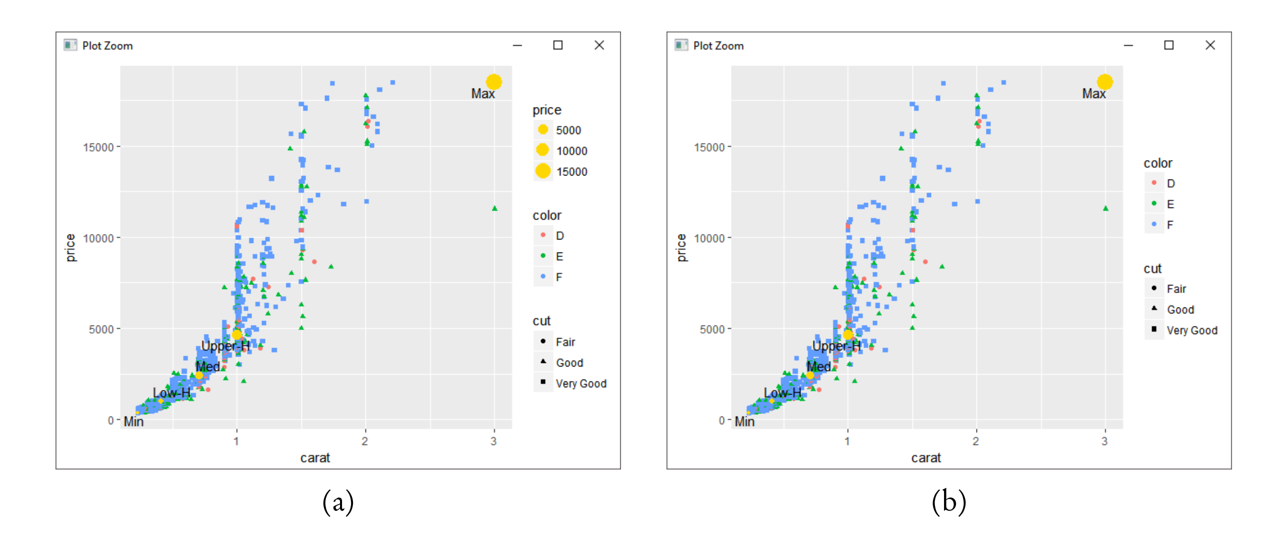
Figure 2.42 – Modification de l'apparence de la légende

Par ailleurs, l'on dispose de la fonction guides() qui opère de la même façon que labs(), elle dispose des paramètres fill, colour…, qui permettent avec la fonction guide_legend(ncol=, byrow=FALSE, reverse = FALSE, title=) de définir l'apparence de la légende. Nous avons également guide_colourbar() qui est l'équivalent de guide_legend() pour les aesthetiques dépendant de variables continues.

Suppression de la légende ou des éléments de la légende

Pour supprimer la légende, il suffit de faire avec **theme()**, **legend.position** = **"none"** ou à travers la fonction **guides()** définir à l'aesthetique correspondant comme **FALSE** par exemple **fill**=**FALSE**. Il peut s'agir de supprimer juste le titre dans la même fonction **guides()** on peut définir **title**=**NULL**.
Ce qui peut sembler intéressant c'est parfois, la possibilité de pouvoir définir des esthétiques et ne pas vouloir que ces derniers paraissent dans la légende. Pour se faire, on utilise l'argument **show.legend**=**FALSE** des fonctions **geom_xxx()**:

```
> df <- subset(diamonds,
      cut == c("Fair","Good","Very Good") &
      color == c("D" ,"E" ,"F"))
> p <- ggplot(df)+ geom_point(aes(carat,price,
                  colour = color, shape = cut))
> p + geom_point(data = data.frame(carat = fivenum(df$carat),
                                  price = fivenum(df$price)),
           aes(x = carat, y = price,size=price),
             colour = "gold") +
 geom_text_repel(data = data.frame(carat = fivenum(df$carat),
                                  price = fivenum(df$price)),
           aes(x = fivenum(carat),y = fivenum(price)),
             label= c("Min", "Low-H","Med","Upper-H","Max"))
> #supprimer la legende relative aux fives numbers
> p + geom_point(data = data.frame(carat = fivenum(df$carat),
                                  price = fivenum(df$price)),
              aes(x = carat, y = price,size=price),
              colour = "gold",show.legend = FALSE) +
  geom_text_repel(data = data.frame(carat = fivenum(df$carat)
    ,
                                  price = fivenum(df$price)),
                 aes(x = fivenum(carat),y = fivenum(price)),
             label= c("Min", "Low-H","Med","Upper-H","Max"))
```

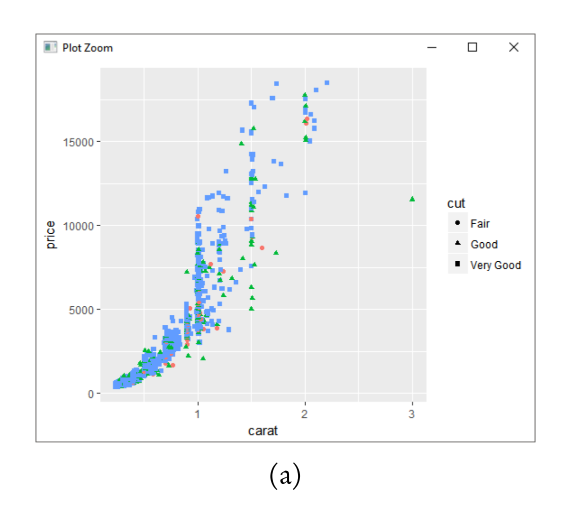


(a) (b)

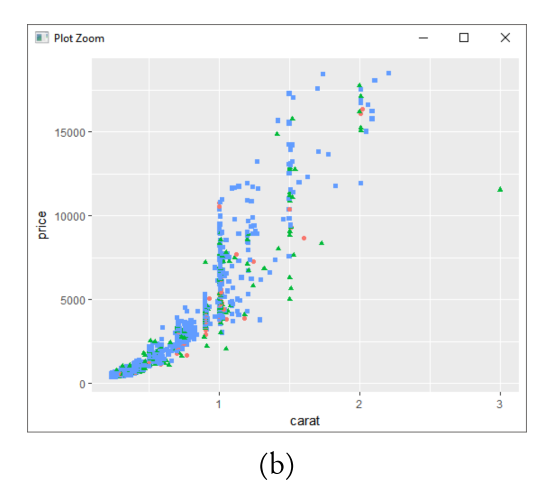
Figure 2.43 – Suppression ou omission des éléments de la légende

Aussi, la fonction **guides()**, permet de supprimer une légende spécifique à une ou plusieurs aesthétiques de notre choix, il suffit pour cela de définir ces aesthétiques comme **FALSE** dans la fonction **guides()** (voir le code ci-dessous):

```
# supprimer la légende de colour seulement
p + guides(colour=FALSE)
# supprimer toute la légende
p + theme(legend.position = "none")
```



(a)



(b)

Figure 2.44 – Suppression totale ou partielle de légende

2.4 Quelques dernières précisions

La fonction **qplot()** et **ggplot()**

Nous avons commencé avec **qplot()** et nous avons entamé ensuite **ggplot()**. Pour cette dernière nous avons démontré une multitude de couches ou layers à ajouter pour permettre un niveau de personnalisation élevée. Et ça pourrait laisser penser au lecteur que **qplot()** est moins que **ggplot()**. Mais ce n'est pas vrai structurellement et dans faits, **qplot()** et **ggplot()** génère les mêmes objets la différence est que **qplot()** a été conçue pour aller vite ! Ainsi nous pouvons ajouter des couches et layers aux objets générés par la fonction **qplot()** et **ggplot()** :

```
> ##
> ggplot(diamonds, aes(x = carat, y = price)) + geom_point()
> # équivalent à
> qplot(x = carat, y = price, data = diamonds)
> ### En ajoutant des couches ou layers
> # les trois commandes suivantes produisent les mêmes
> # graphiques
> qplot(x = carat, y = price, data = diamonds) +
    geom_smooth()
> # ou
> qplot(x = carat, y = price, data = diamonds,
                geom = c("point", "smooth"))
> # ou
> ggplot(diamonds, aes(x = carat, y = price)) +
```

```
                geom_point() + geom_smooth()
```

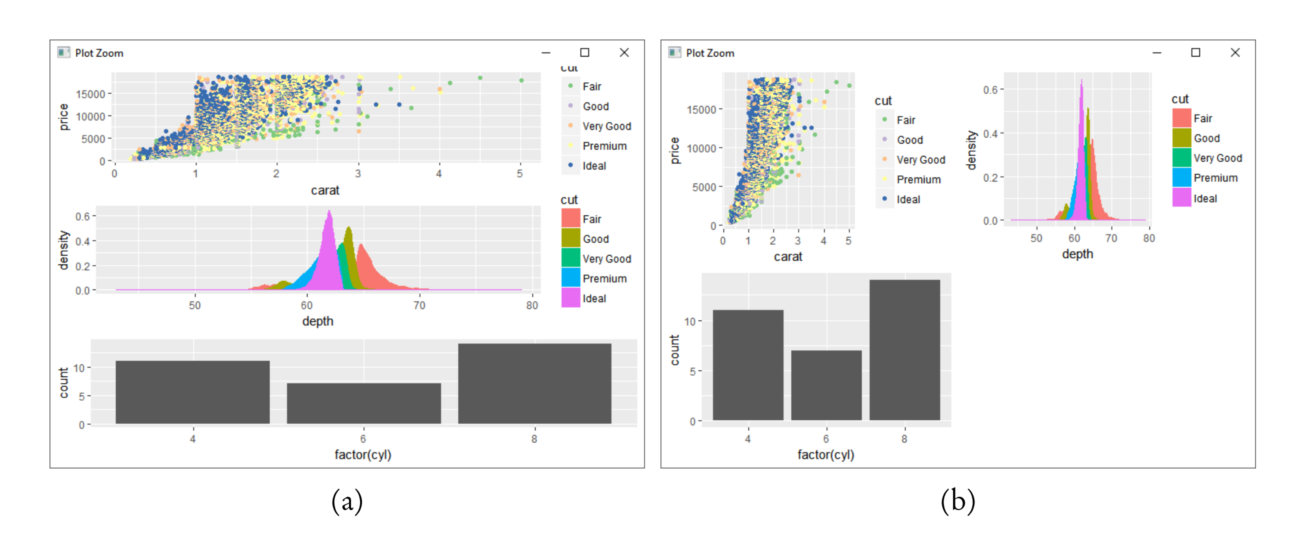
Combiner plusieurs graphiques sur une seule figure

Les objets **ggplot2** ne dépendant pas des objets graphiques de base R, la fonction **par()** ou **layout()** ne peut être utilisée pour découper la figure en multiple zone graphique permettant d'accueillir plusieurs graphiques. **ggplot2** tire ces sources des objets **grid** donc pour combiner plusieurs graphiques en une figure graphique nous avons les possibilités suivantes :

La fonction grid.arrange() du package Extragrid

Précisons que **Extragrid** est une extension du package **grid** donc, il permet également de manipuler des objets **ggplot2**.
Avec la fonction **grid.arrange()**, l'on peut ranger les graphiques à sa guise, notamment avec les paramètres **nrow** et **ncol** qui permettent de ranger les graphiques comme on peut le voir sur la figure 2.45 dont voici le code ci-dessous :

```
> library(gridExtra)
> p1 <- qplot(data=diamonds, x= carat,y = price,
      colour = cut)+
     scale_color_brewer(palette = "Accent")
> p2 <- qplot(x = depth ,data=diamonds, geom="density",
             colour = cut, fill=cut)
> p3 <- ggplot(mtcars,aes(x = factor(cyl)))+geom_bar()
> grid.arrange(p1,p2,p3)
> grid.arrange(p1,p2,p3,nrow=2)
```



(a) (b)

Figure 2.45 – Ranger plusieurs graphiques

Mieux, le paramètre **layout_matrix** permet de définir une matrice à la manière de **layout()** pour découper la figure graphique(voir 2.46)

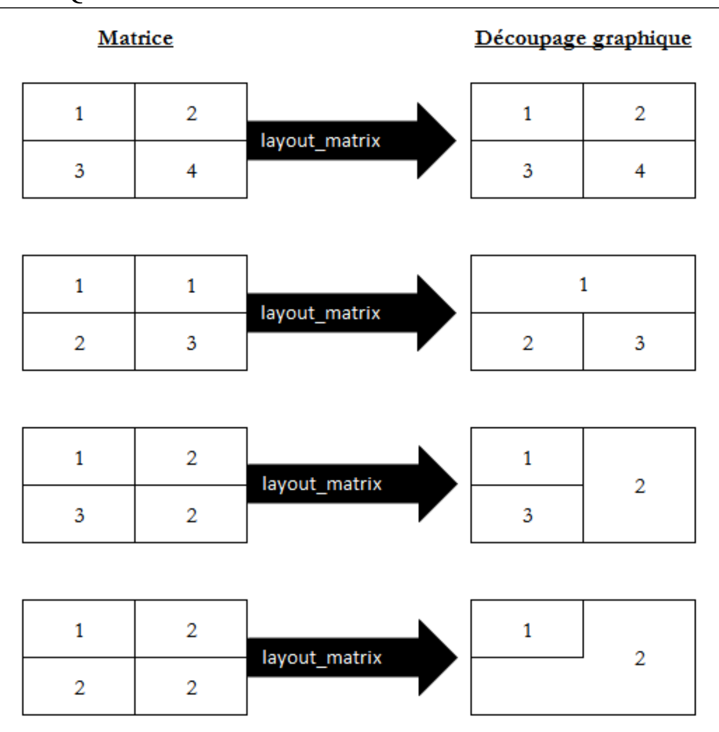
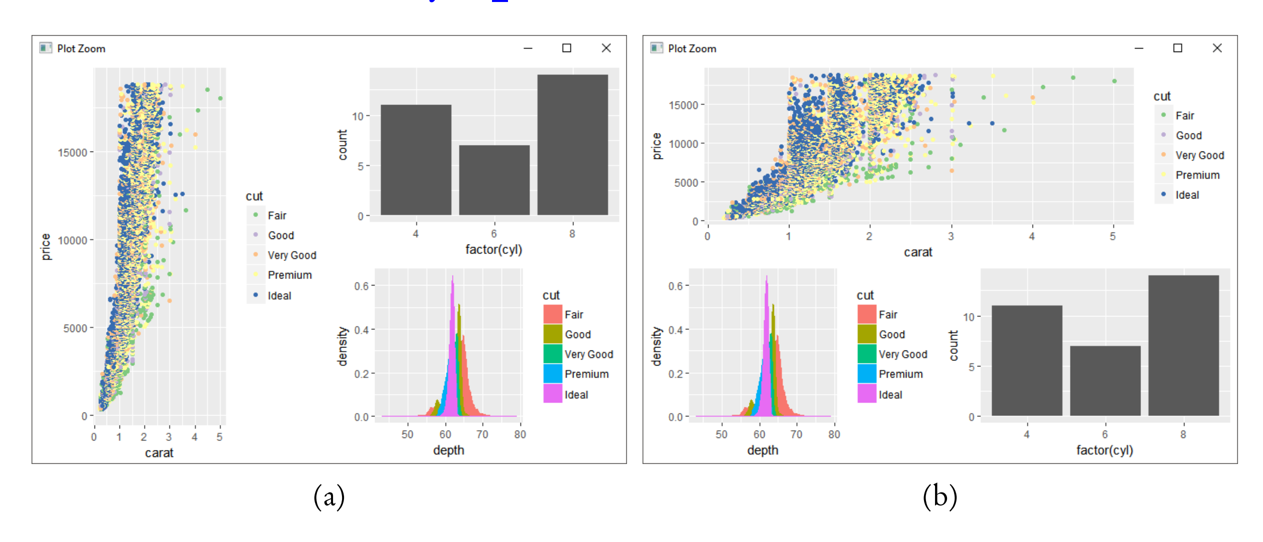


Figure 2.46 – Illustration du fonctionnement de layout_matrix

```
> grid.arrange(p1,p2,p3,
            layout_matrix = cbind(c(1,1),
                                  c(3,2)))
> grid.arrange(p1,p2,p3,
            layout_matrix = cbind(c(1,2),
                                  c(1,3)))
```

Ainsi, les graphiques 2.47a et 2.47b présentent les mêmes graphiques selon différent rangement de la matrice fournie à **layout_matrix**.



(a)

(b)

Figure 2.47 – Ranger plusieurs graphiques, paramètre **layout_matrix**

On peut même faire partager les mêmes titres avec le paramètre **top**, **bottom**, **left**, ou **right** (selon l'endroit où l'on veut positionner le titre, par ailleurs ces paramètres ne se substituent

pas, ce sont en fait des outils d'annotations qui peuvent prendre **textGrob()** en valeur) de la fonction **grid.arrange()** ou les mêmes légendes aux graphiques combinés :

```
> library(grid)
> p4 <- qplot(x = depth ,data = diamonds, geom = "histogram",
          colour = cut, fill = cut)
> p5 <- qplot(x = cut,y= carat,data = diamonds, geom = "boxplot",
               colour = cut)
> plots <- list(p2,p4,p5)
> g <- ggplotGrob(plots[[1]] +
             theme(legend.position="bottom"))$grobs
> # extraction de la légende
> legend <- g[[which(sapply(g,function(x) x$name) == "guide-box")]]
> lheight <- sum(legend$height)
> grid.arrange(
    # extraction des objets grobs de graphique
    # avec omission de la légende
    do.call(arrangeGrob, lapply(plots, function(x)
     x + theme(legend.position="none"))),
     # ajout de text avec textGrob() qui permet l'ajout
     # de style
     top = textGrob("Un titre pour tous !",
     gp = gpar(fontsize=15,col = "red3"),vjust =1),
     # ajout de texte sans style à gauche
    left = "Un text à gauche",
    legend, # ajout de la légende
    ncol=1,
    heights = unit.c(unit(1, "npc") - lheight, lheight))
```

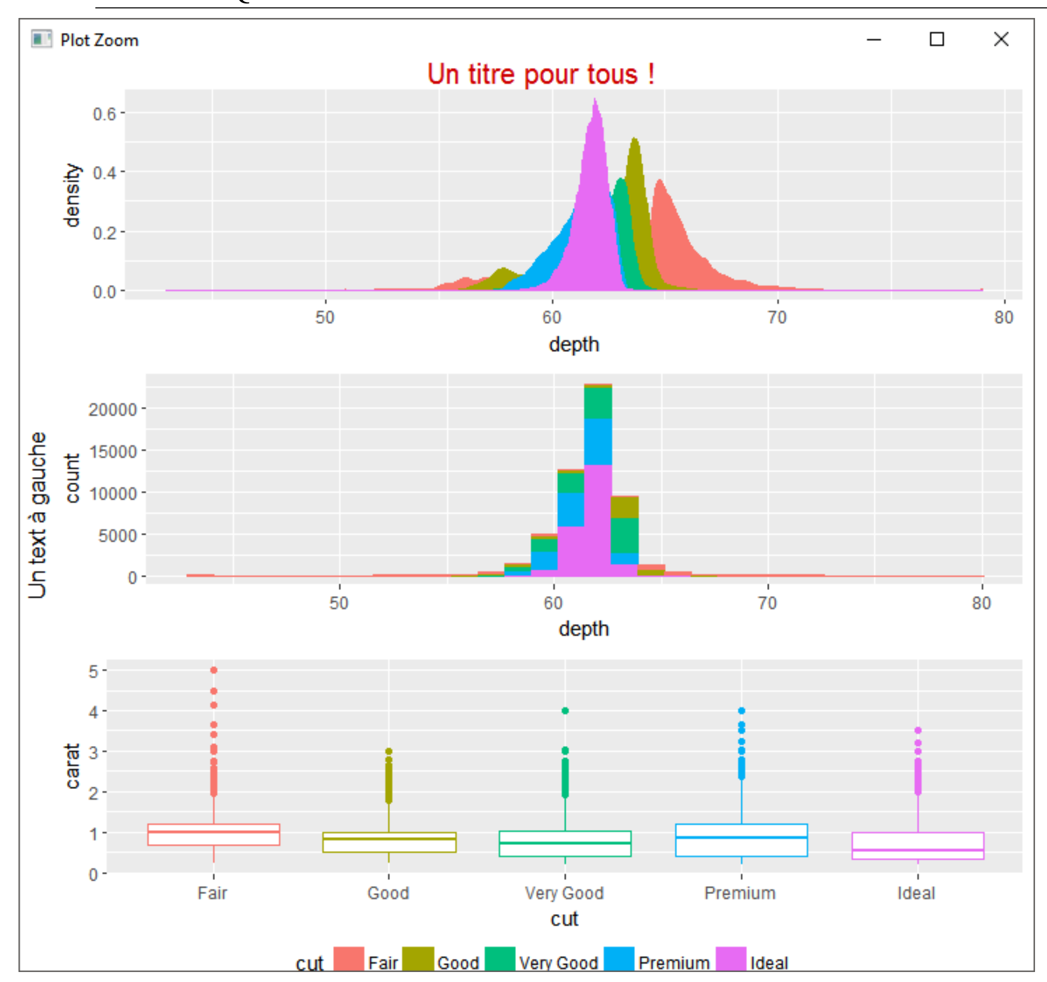


Figure 2.48 – Légende et titre partagés pour des graphiques combinés

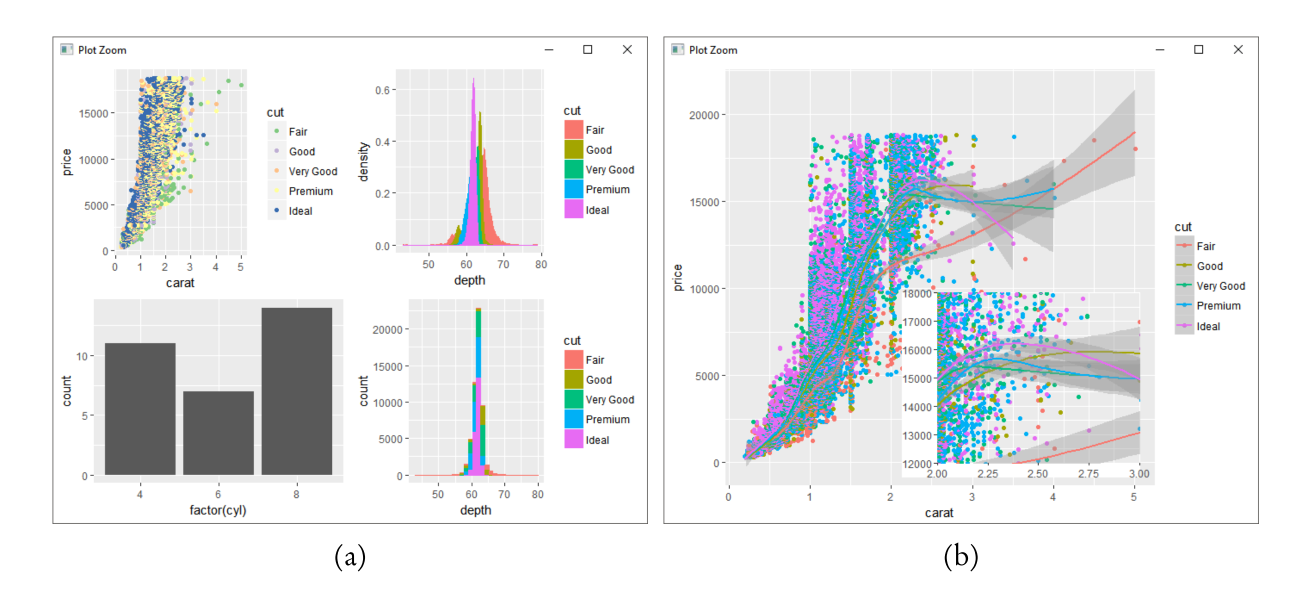
Pour comprendre au mieux les objets **grid** et **gtable** en fin de contrôler avec efficacité les objets **gplot2** voici un lien intéressant Site

La fonction viewport() de grid

Un objet **viewport** permet de présenter un objet géométrique relatif au package **grid**. Ainsi, toute la figure graphique est un **viewport** et avec son paramètre **layout**, on peut découper à notre guise la figure graphique. Ensuite, nous utilisons la fonction **pushViewport()** pour projeter le **viewport()** une fois le layout définit. Les paramètres **layout.pos.row** et **layout.pos.col** permettent de sélectionner chaque subdivision ou zone graphique selon son rang ligne et colonne. Avec la fonction **print()** on peut afficher des objets **gplot** :

```
> # nouvelle figure graphique
> grid.newpage()
> # definir les layout ou le decoupage
> pushViewport(viewport(layout = grid.layout(nrow=2, ncol=2))
  )
> print(p1, vp = viewport(layout.pos.row = 1, layout.pos.col
  = 1))
> print(p2, vp = viewport(layout.pos.row = 1, layout.pos.col
  = 2))
```

```
> print(p3, vp = viewport(layout.pos.row = 2, layout.pos.col
   = 1))
> print(p4, vp = viewport(layout.pos.row = 2, layout.pos.col
   = 2))
```



(a) (b)

Figure 2.49 – Ranger plusieurs graphiques avec **viewport()**

Par ailleurs, nous pouvons également faire des projections spécifiques et relatives à un viewport préexistant comme présenté dans le code ci-dessous (voir le résultat sur la figure 2.49b)

```
p1 <- ggplot(diamonds)+ aes(carat,price,colour = cut) +
      geom_point() + geom_smooth()
print(p1, vp = viewport(width = 1,
                     height = 1, x=0.5, y = 0.5))
# zoom une partie spécifique
p2 <- p1 + coord_cartesian(xlim = c(2, 3),
             ylim=15000+c(-3000,3000)) +
      scale_x_continuous(expand = c(0,0)) +
      scale_y_continuous(expand = c(0,0))+
      theme(plot.margin = unit(c(0,0,0,0),"npc"),
            legend.position = "none",
            axis.title = element_blank())
print(p2, vp = viewport(width = 0.4, height = 0.4,
                     x=0.6, y = 0.3))
```

Sauvegarde des graphiques

Pour sauvegarder nos graphiques dans le système **ggplot2**, nous avons la fonction **ggsave()** qui offre d'autres options sur la définition des dimensions du graphique à produire :

```
> ggplot(mtcars, aes(mpg, wt)) + geom_point()
> # format pdf
> ggsave("mtcars.pdf")
> # format png
```

```
> ggsave("mtcars.png")
> # redimensionnement
> ggsave("mtcars.pdf", width = 4, height = 4)
> ggsave("mtcars.pdf", width = 20, height = 20, units = "cm")
> # format jpeg
> ggsave("mtcars.jpg", width = 4, height = 4)
```

Naturellement dans le code ci-dessus, c'est le graphique actuel **last_plot()**, qui sera sauvegarder. On peut spécifier explicitement l'objet graphique à sauvegarder à travers le paramètre **plot** qui par défaut est égal à **last_plot()**(qui désigne le dernier objet graphique **ggplot** affiché).
Bien entendu comme on peut le voir dans le code ci-dessus, la fonction **ggsave()** supporte plusieurs format de fichier, des images, pdf. . . Les autres fonctions habituelles telles que **png()** ou **pdf()** ou **jpeg()**. . . sont fonctionnelles également pour les graphiques **ggplot2**.

3

Galerie graphique

Contents

Ce chapitre se consacrera uniquement à présenter les différents types de graphique les plus usuels avec **ggplot2**. Les données qui seront utilisées pour ce fait, sont celles du **GapMinder** et les packages utilisés sont notamment :

- **dplyr** et **plyr** pour la trituration des données
- **scales** pour le formatage des données,
- **ggrepel** pour l'annotation graphique notamment les nuages de points.
- **grid** pour modifier l'arrangement des graphiques

Avant de commencer, nous allons d'abord télécharger les packages précités et les données.

```
> load('gapdata.RData') # chargement des données
> str(gapdata)
'data.frame':   1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1
```

```
 $ year      : int  1952 1957 1962 1967 1972 1977 1982 1987
 $ pop       : num  8425333 9240934 10267083 11537966  ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
> # chargement des librairies
> library(ggplot2)
> library(ggprepel)
> library(plyr)
> library(dplyr)
> library(scales)
> library(grid)
```

3.1 Diagrammes X-Y et à lignes

Diagrammes X-Y avec des marques mineures aux axes des diagrammes X-Y et double axes des ordonnées

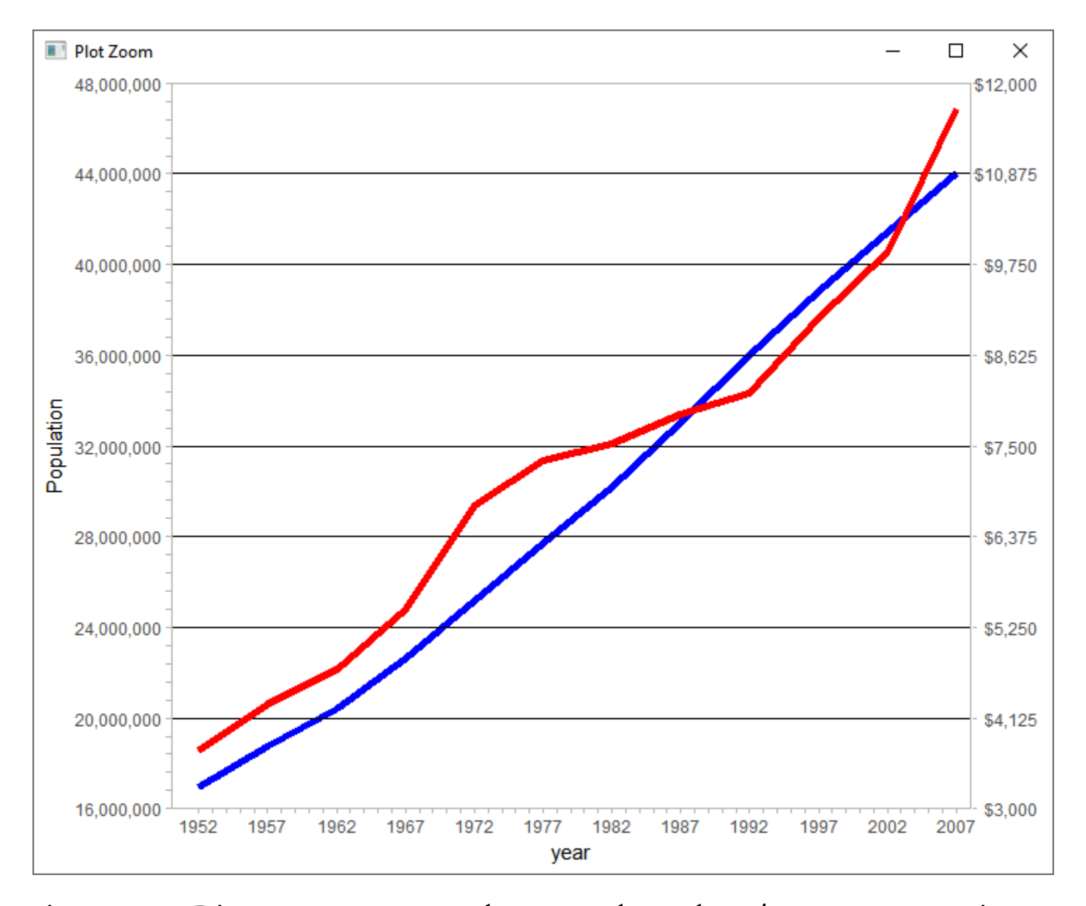


Figure 3.1 – Diagramme X-Y avec deux axes des ordonnées et marques mineurs

Diagrammes X-Y avec Ajout des légendes aux points

Nous avons déjà abordé l'ajout de légendes aux points dans la section annotation du chapitre 2. Toutefois, nous allons faire une illustration avec geom_text_repel() du package ggrepel pour produire le graphique 3.2.

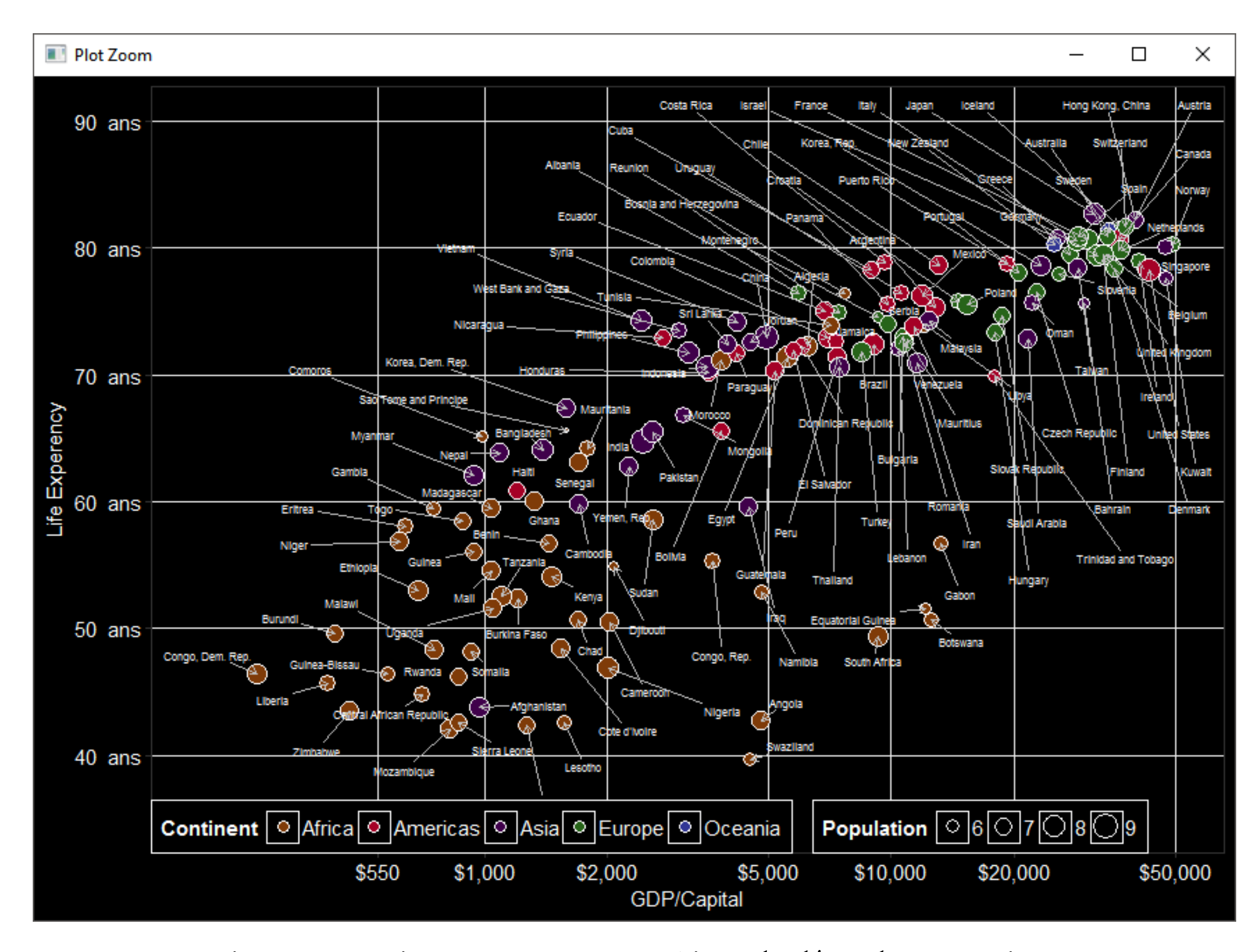


Figure 3.2 – Diagrammes X-Y avec Ajout des légendes aux points

Zoom sur un sous-ensemble des diagrammes X-Y

Pour zoomer sur un sous-ensemble se référer à la réalisation des figures 2.27a, 2.27b et la graphique 2.49b

Diagrammes X-Y avec ajout de barres d'erreurs

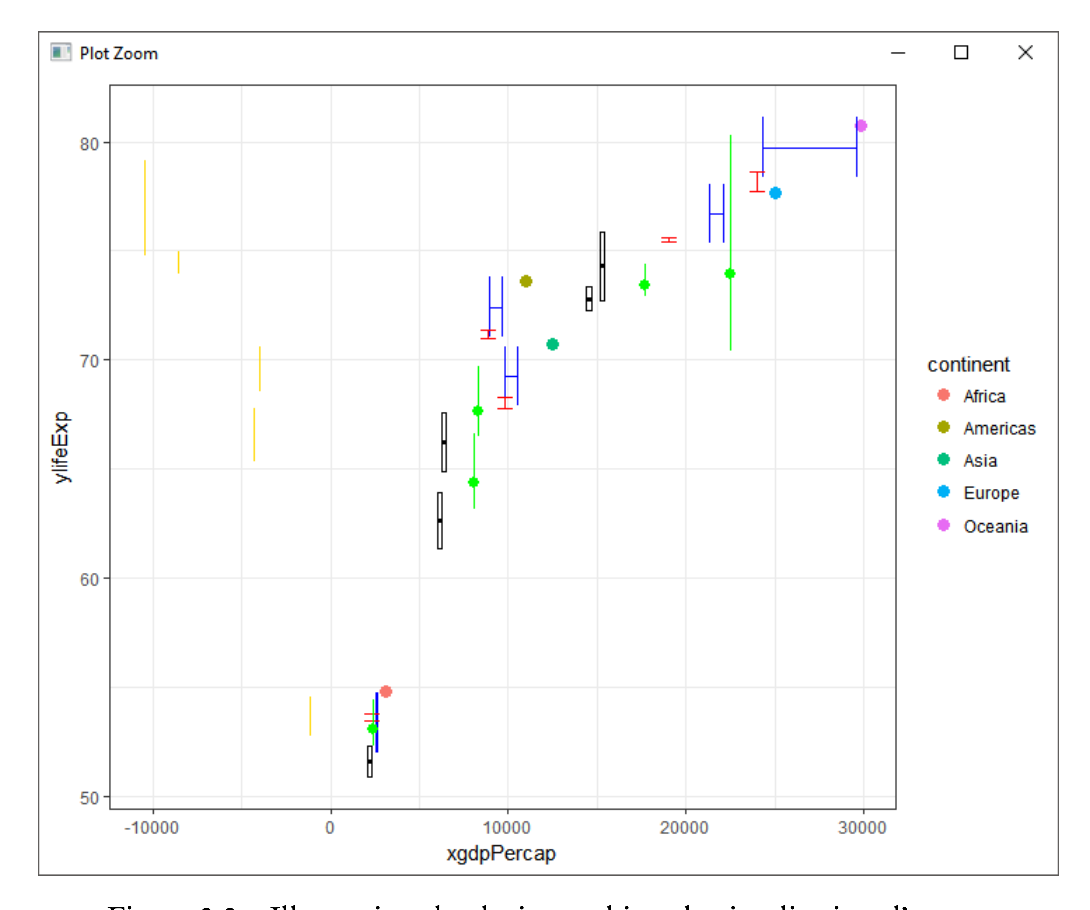


Figure 3.3 – Illustration de plusieurs objets de visualisation d'erreur

3.2 Diagrammes à points catégoriels (diagramme de Cleveland)

Diagramme de Cleveland simple avec vignette

Pour réaliser de simples graphiques de Cleveland, nous avons juste besoin des valeurs et de leur étiquette comme par exemple sur la figure 3.4, on n'y voit des pays en ordonnées et leur PIB (**gdpPercap**) :

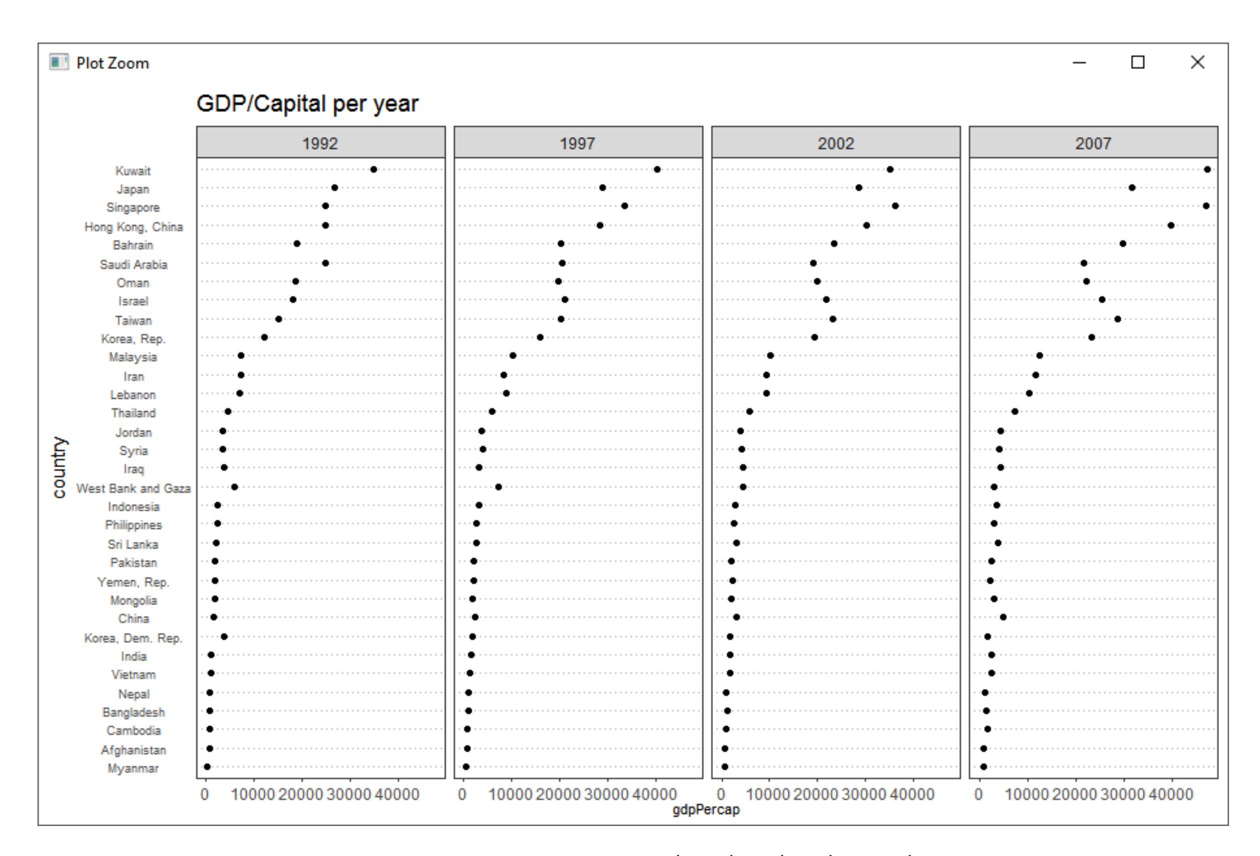


Figure 3.4 – Diagramme de Cleveland simple

Diagramme de Cleveland complexe

Bien attendu il n'existe qu'un seul type de diagramme de Cleveland, nous avons jugé bon toutefois, de présenter deux variantes simple et complexe(ce dernier est souvent appelé graphique en sucette). Le diagramme présenté sur la figure 3.5, diffère du précédent en ce sens que nous allons construire les points par rapport à un indicateur ici la moyenne. Voici comment nous avons préparé les données :

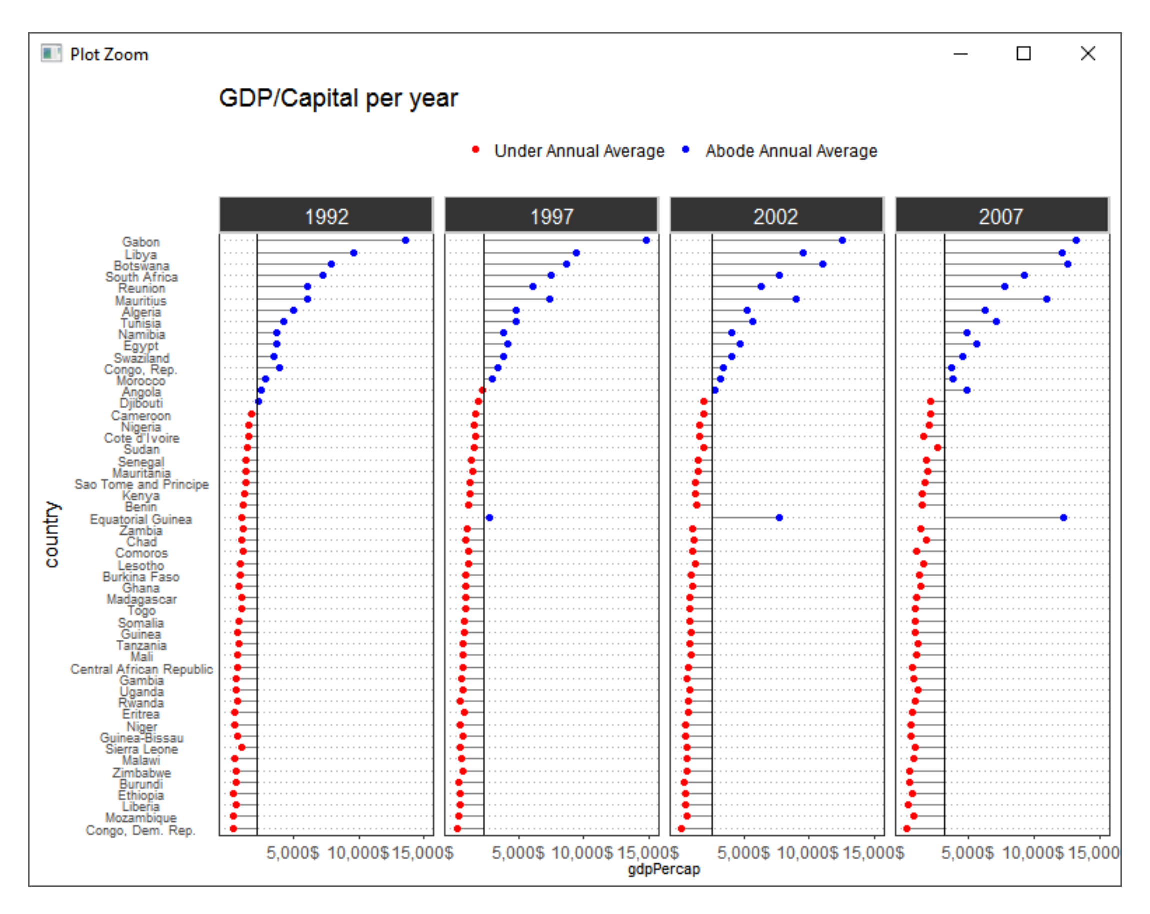


Figure 3.5 – Diagramme de Cleveland complexe avec vignette

3.3 Diagramme à barres

Les diagrammes en barres sont très utiles pour représenter des données sommaires. Avec ggplot2, on utilse geom_bar() ou la version stat_bin() avec spécification du paramètre geom comme vu dans le chapitre 2.

Diagramme à barres superposés avec annotation centrée

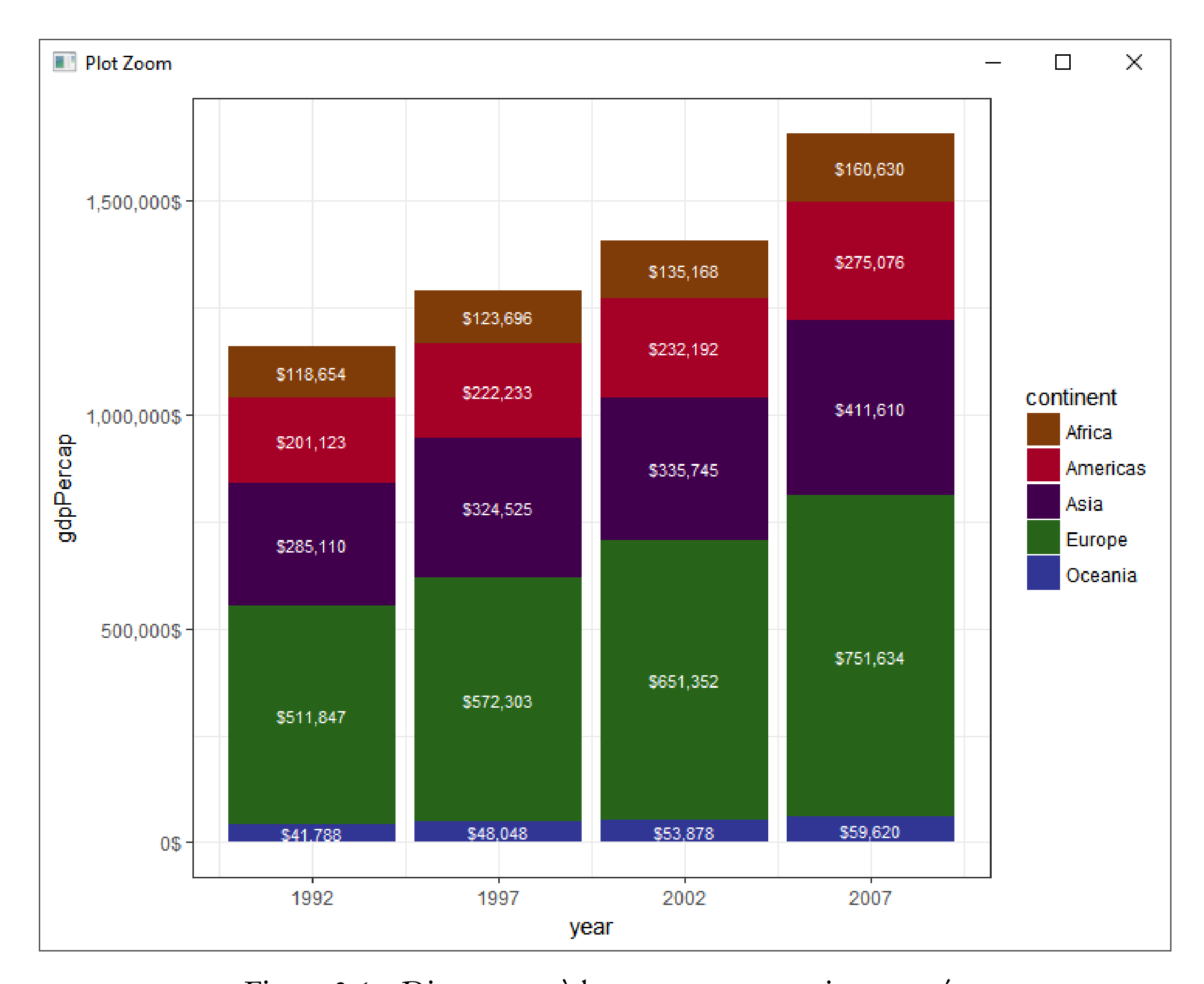


Figure 3.6 – Diagramme à barres avec annotation centrée

Diagramme à barres avec barres d'erreurs et vignettes

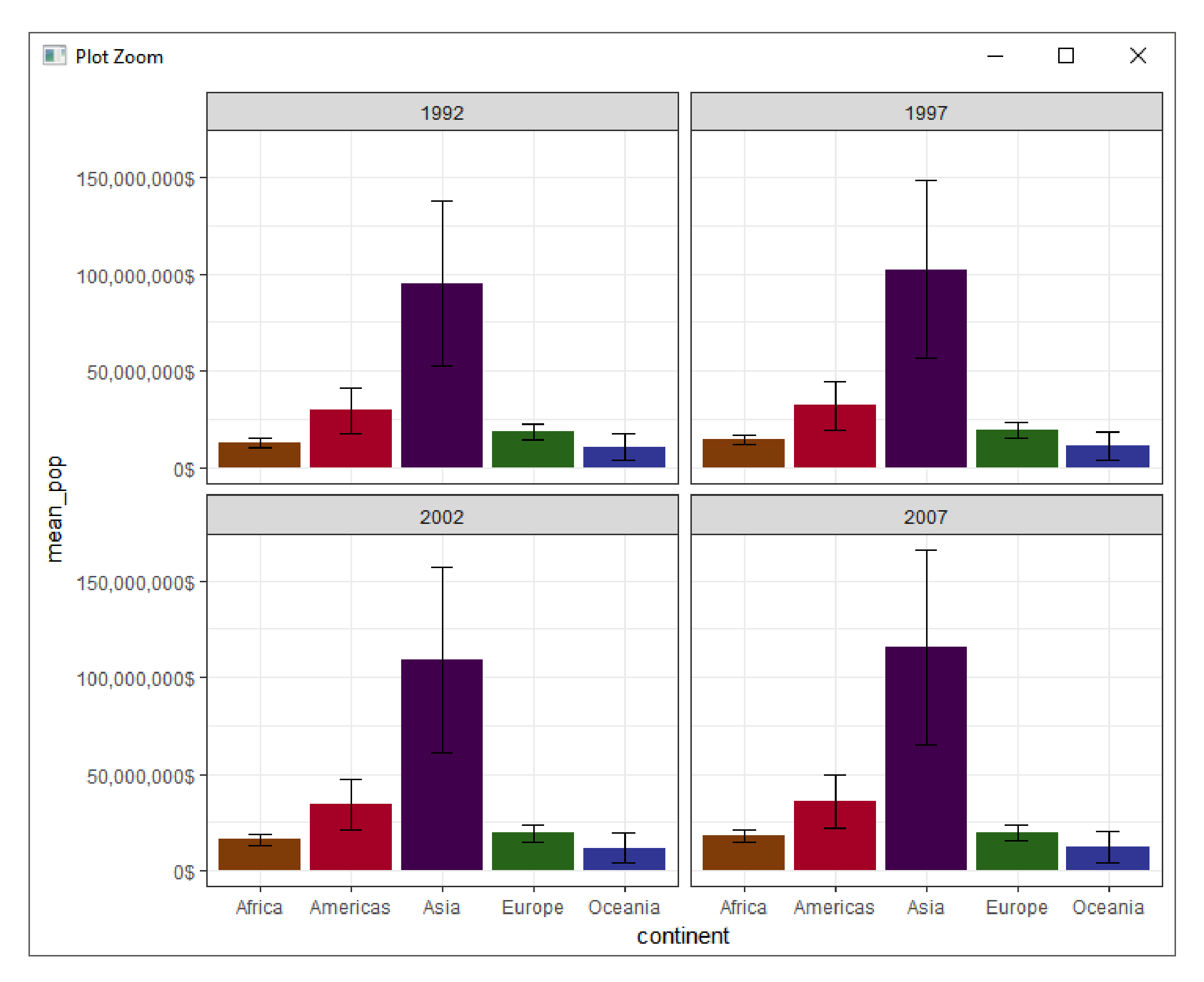


Figure 3.7 – Diagramme à barres avec barres d'erreurs et vignettes

Diagrammes à barres empilées (fréquences)

Pour réaliser la figure 3.8, le gros du travail réside dans la trituration des données en fin d'obtenir les fréquences ou proportions de PIB(**gdpPercap**)pour chaque continent par an :

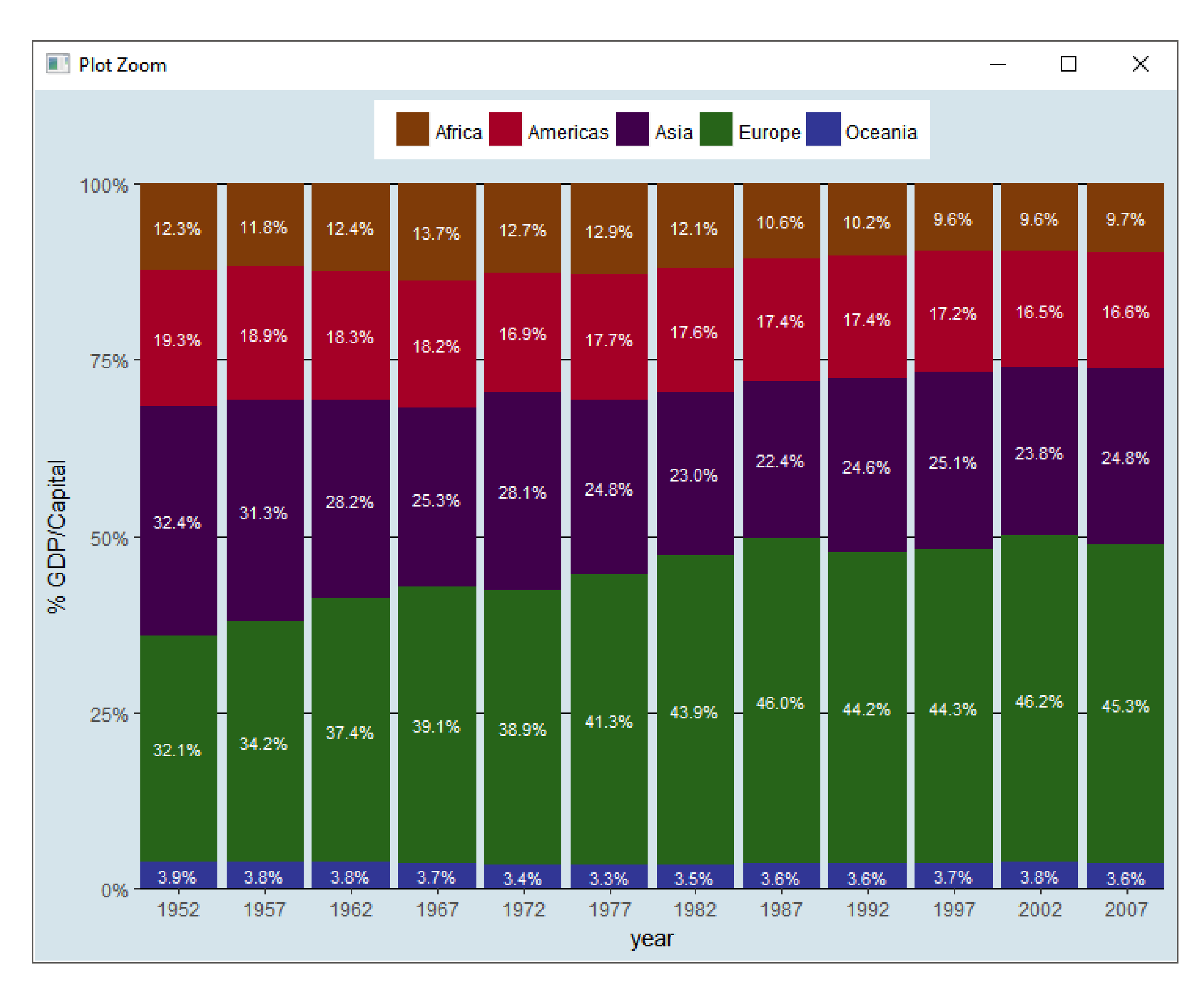


Figure 3.8 – Diagrammes à barres superposés (fréquences) et annotés

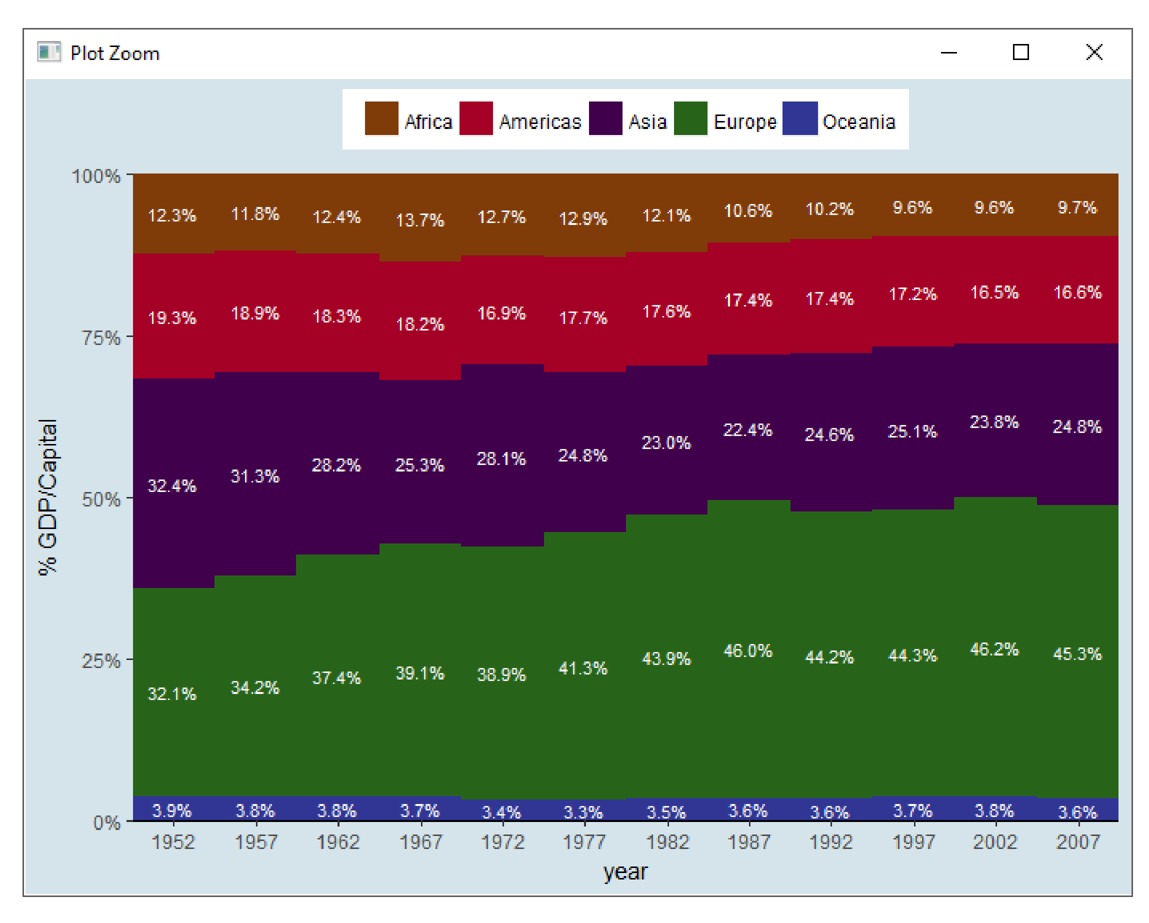


Figure 3.9 – Diagramme à barres avec suppression d'espacement

3.4 Diagramme à bulles

Le graphique 3.10, présente un diagramme à bulles qui est structurellement un nuage de points avec une forme circulaire **shape=21**.

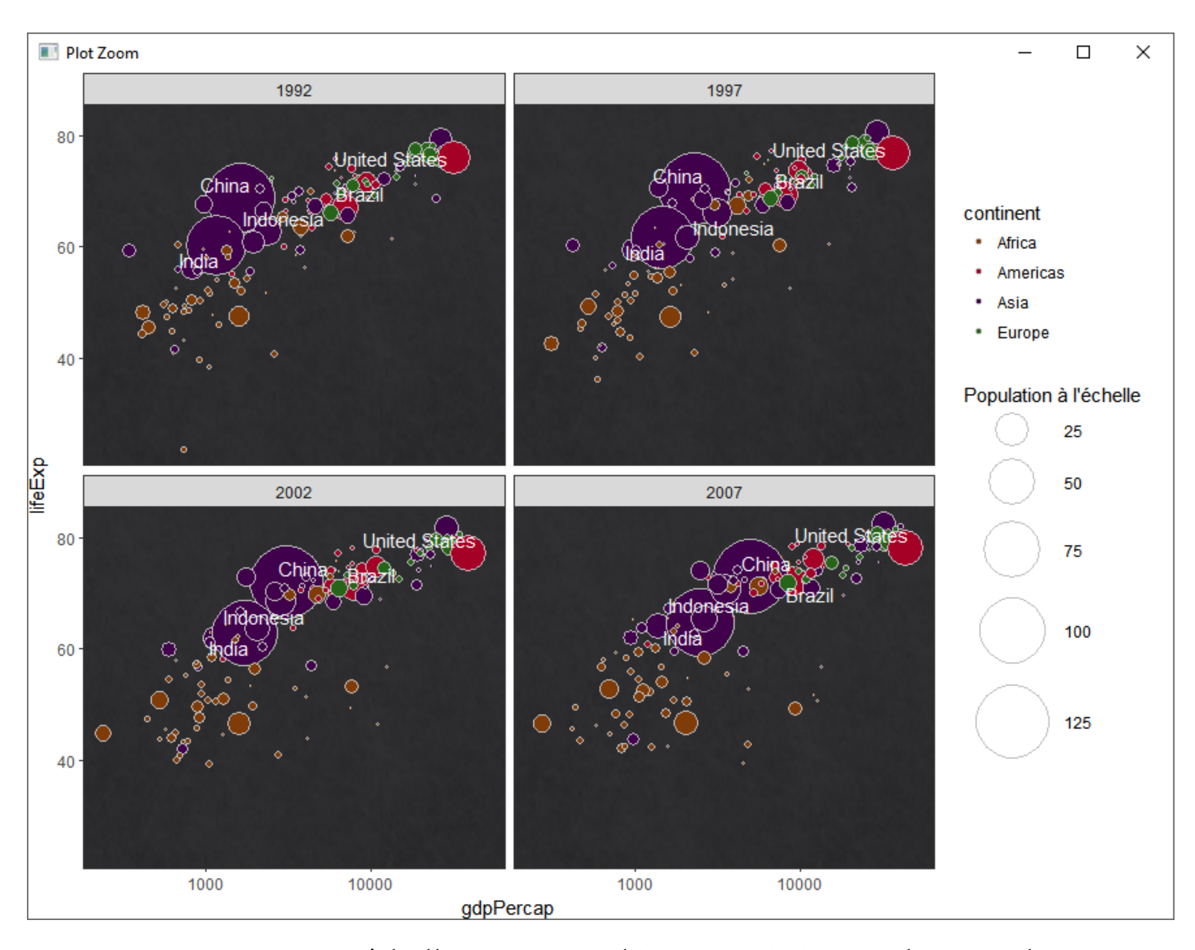


Figure 3.10 – Diagramme à bulles avec ajout d'image statistique et de textes dynamiques

3.5 Diagramme de densité et ses variantes

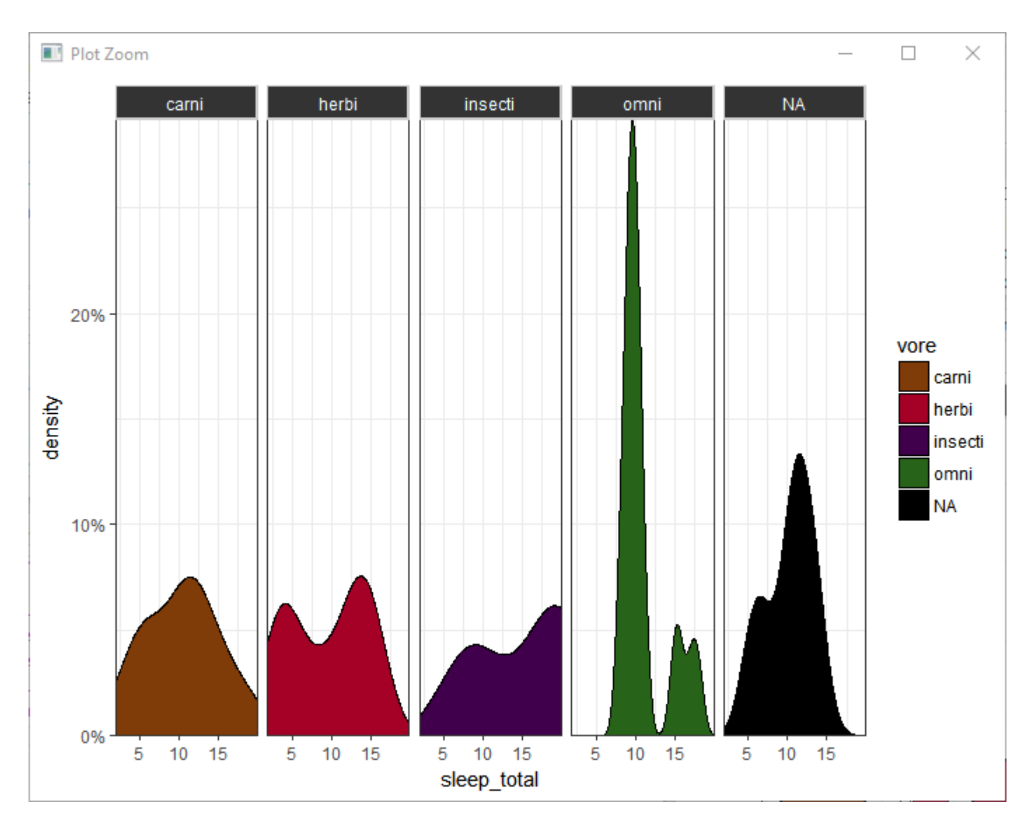


Figure 3.11 – Diagramme de densité avec vignette

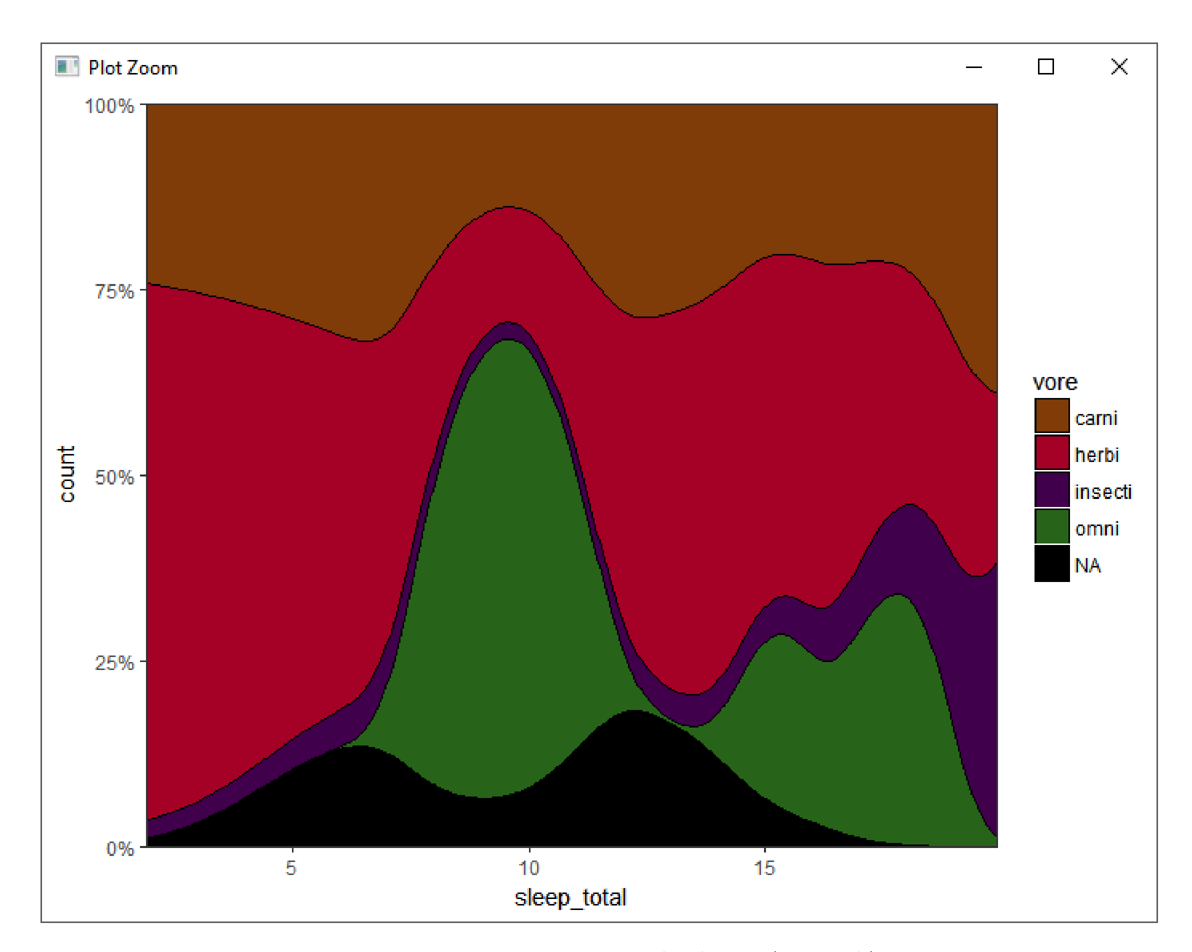


Figure 3.12 – Diagramme de densité empilée

Diagramme avec estimation du kernel de densité en 2D

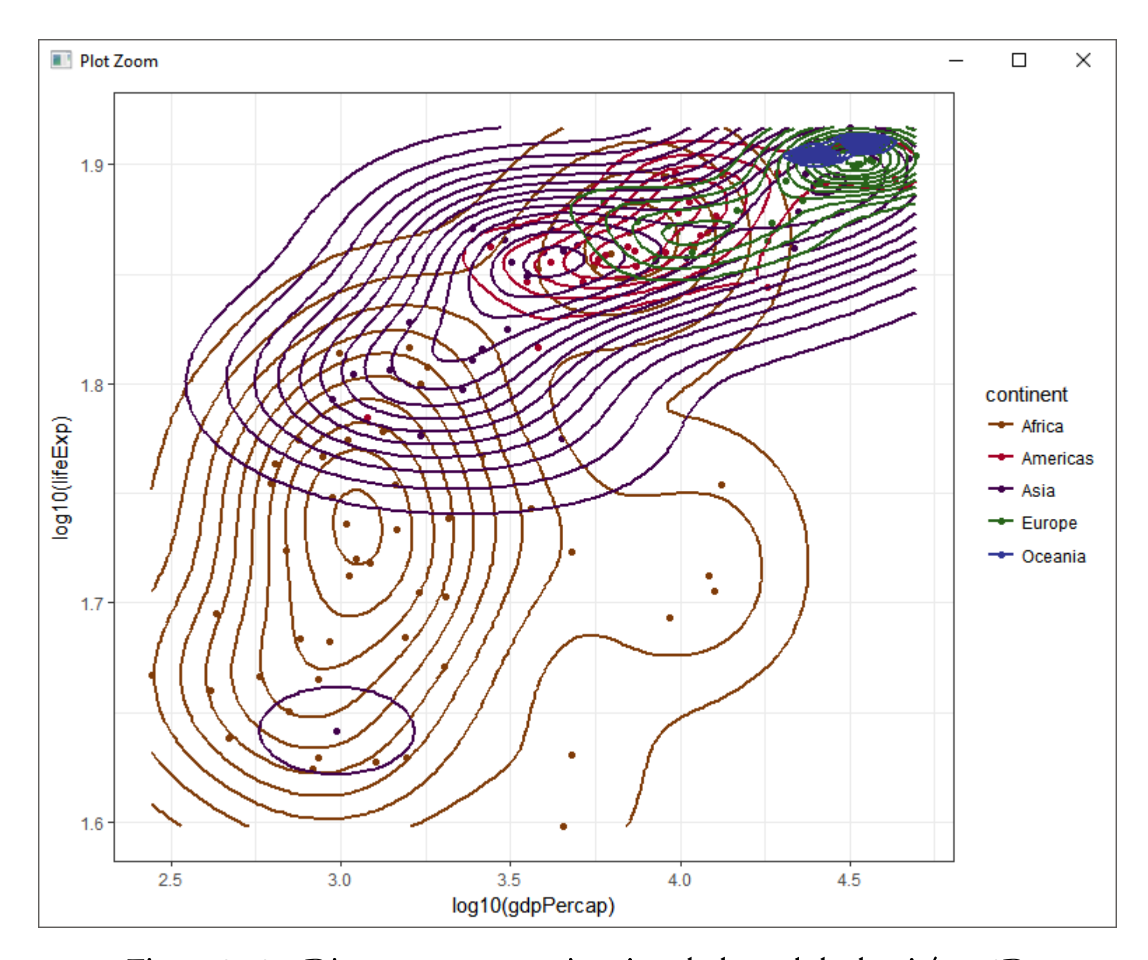


Figure 3.13 – Diagramme avec estimation du kernel de densité en 2D

3.6 Histogrammes

Histogramme (densité) avec des dates

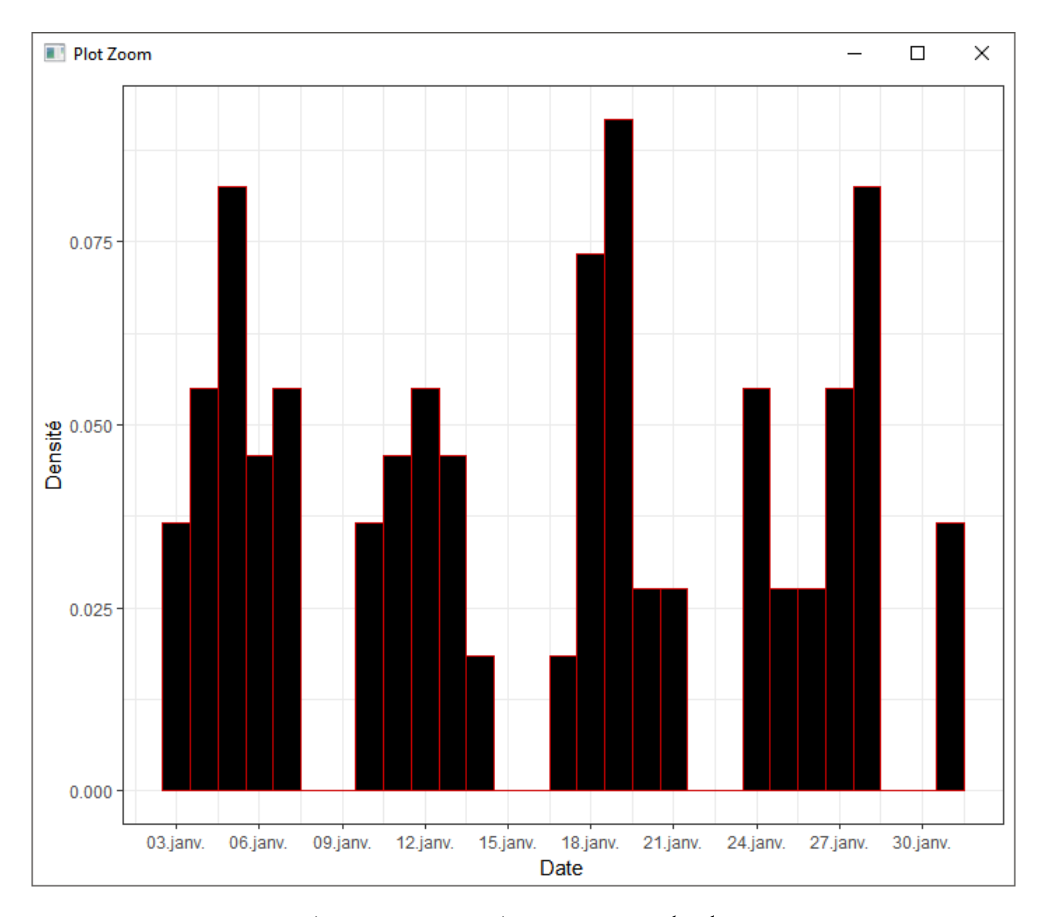


Figure 3.14 – Histogramme de date

3.7 Combinaisons de plusieurs graphiques

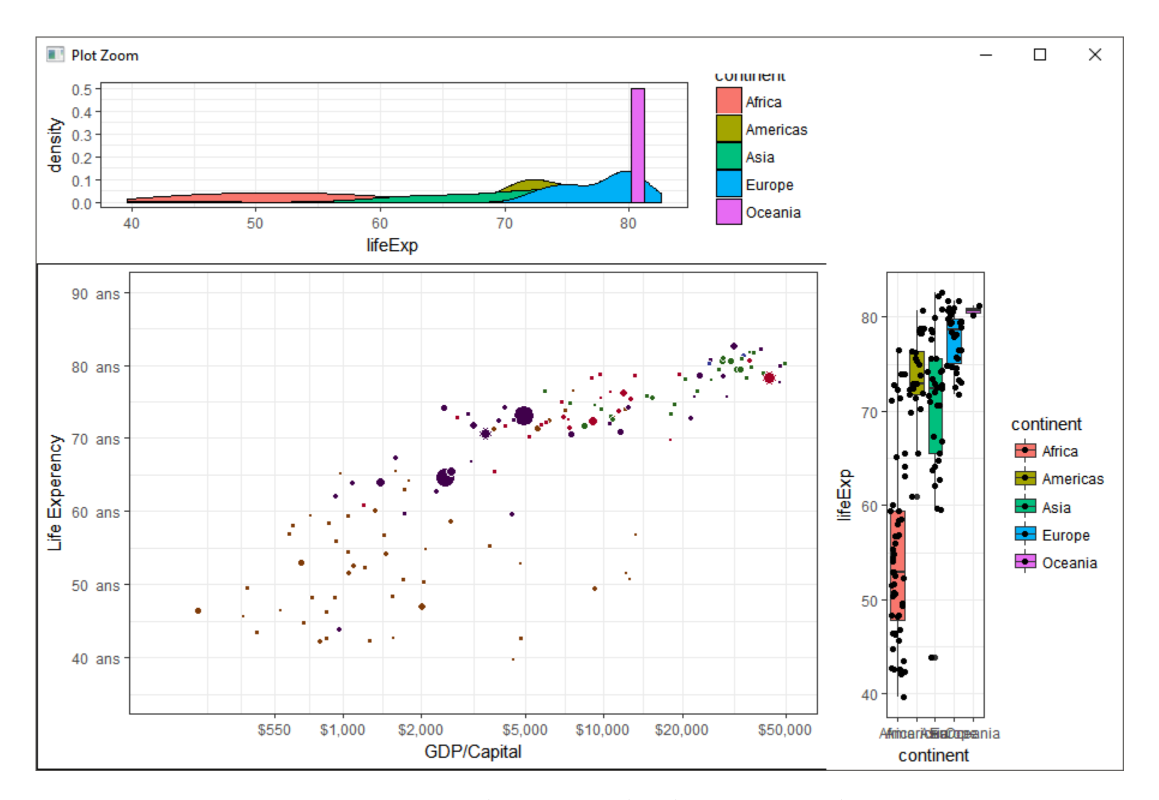


Figure 3.15 – Combinaisons de plusieurs graphiques

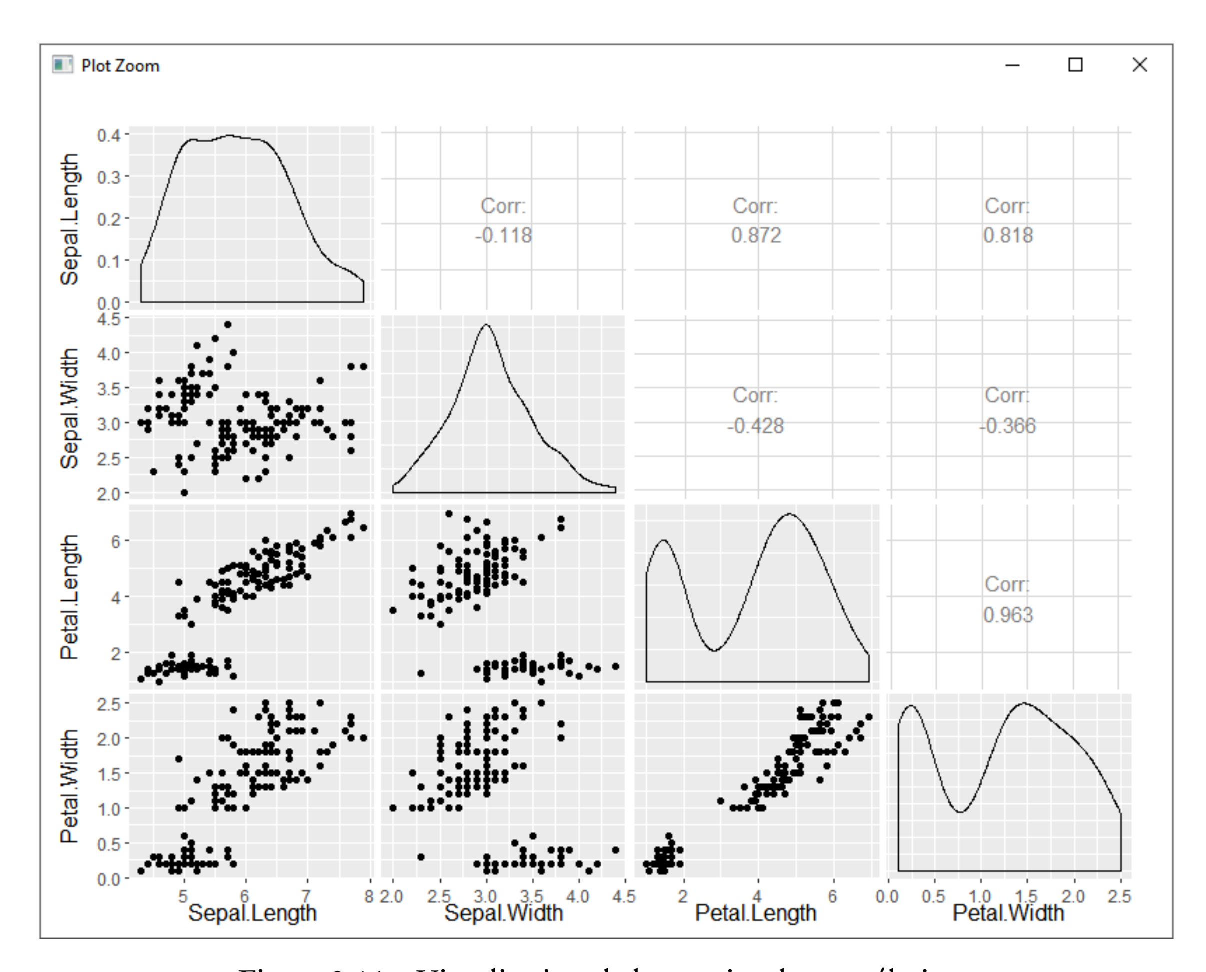


Figure 3.16 – Visualisation de la matrice des corrélations

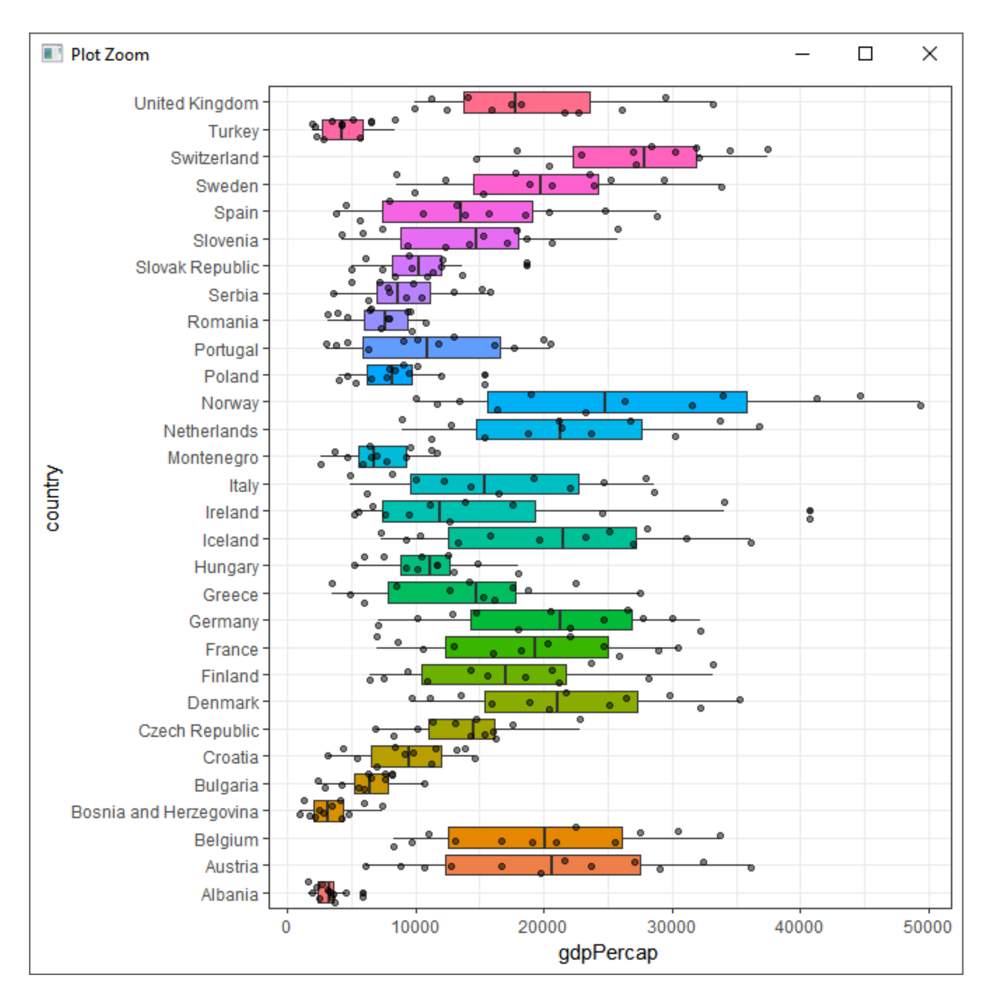


Figure 3.17 – Boîte à moustaches avec rotation des axes et gigue

3.8 Camemberts ou diagrammes à secteur et Donut

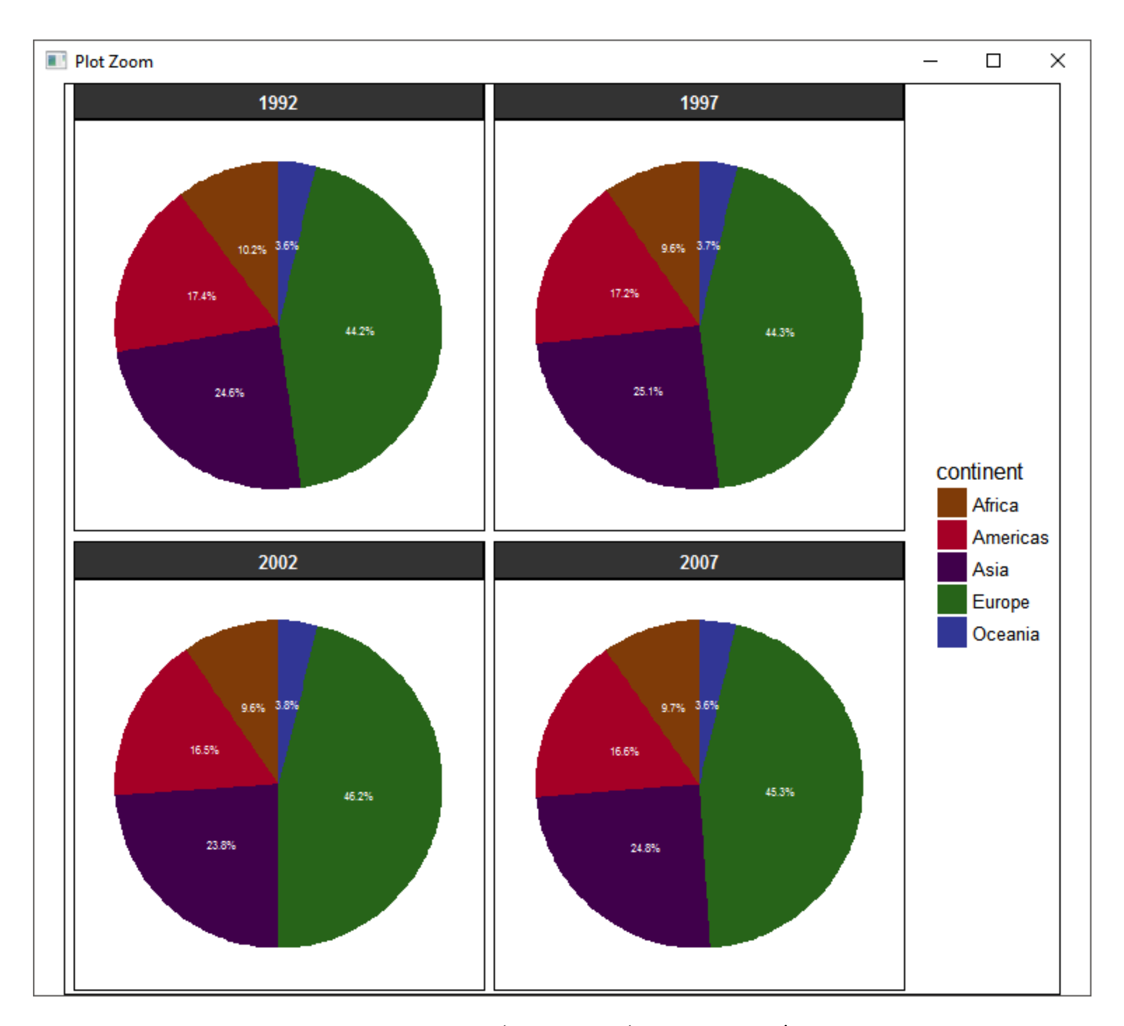


Figure 3.18 – Camemberts ou diagrammes à secteurs

Diagramme Donut

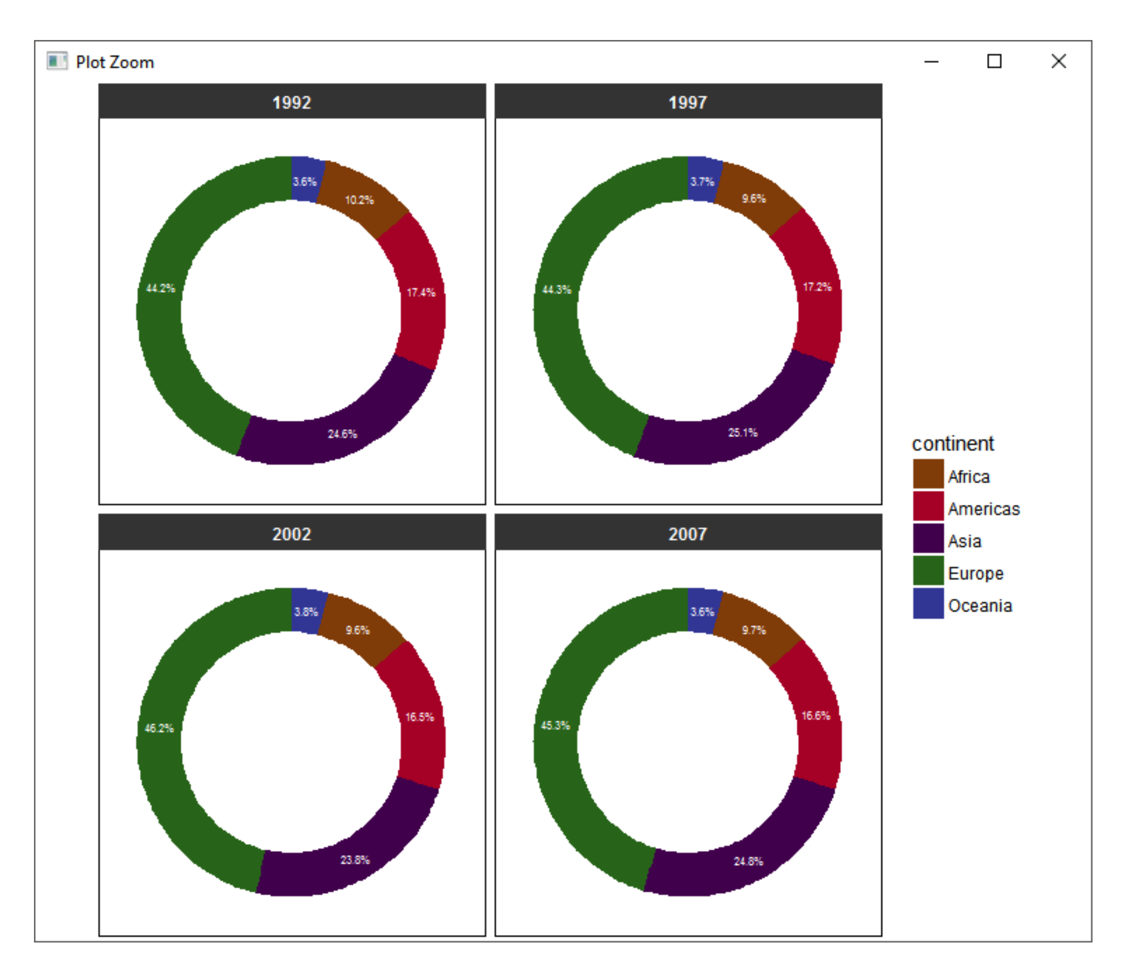


Figure 3.19 – Diagramme Donut

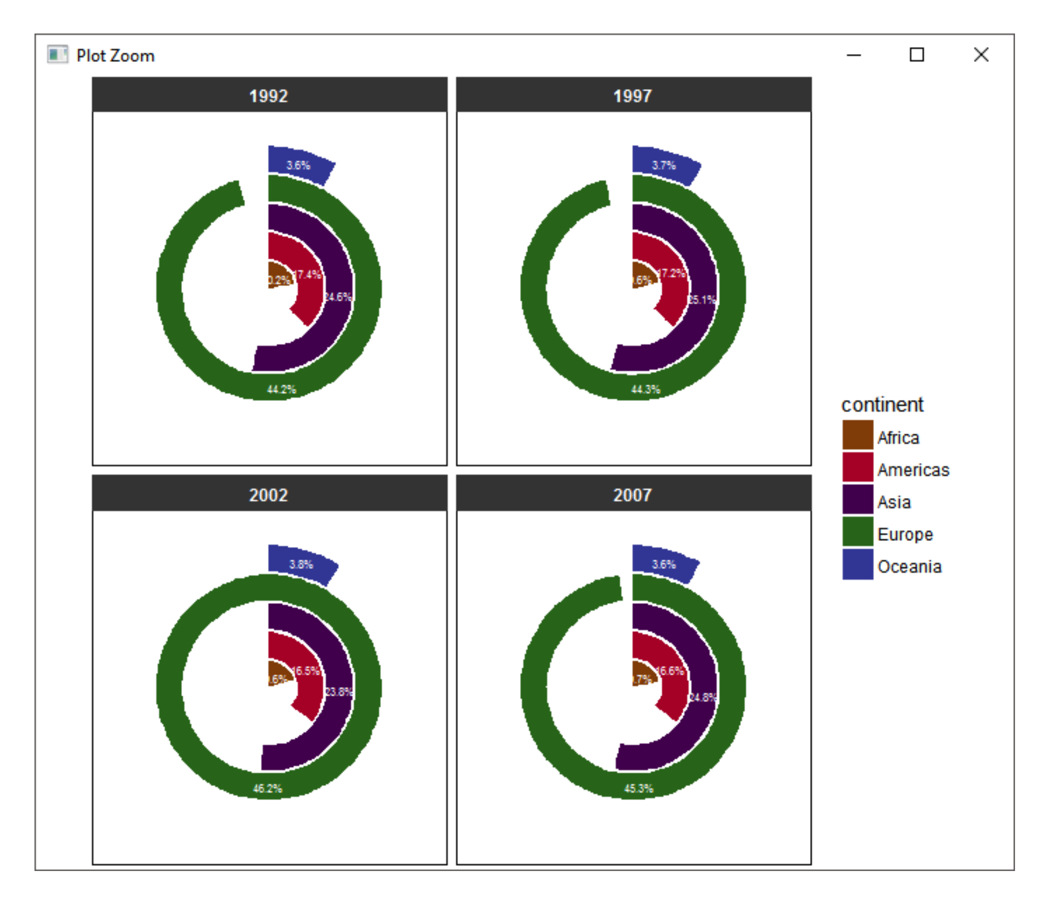


Figure 3.20 – Diagramme en Anneaux

3.9 HeatMap

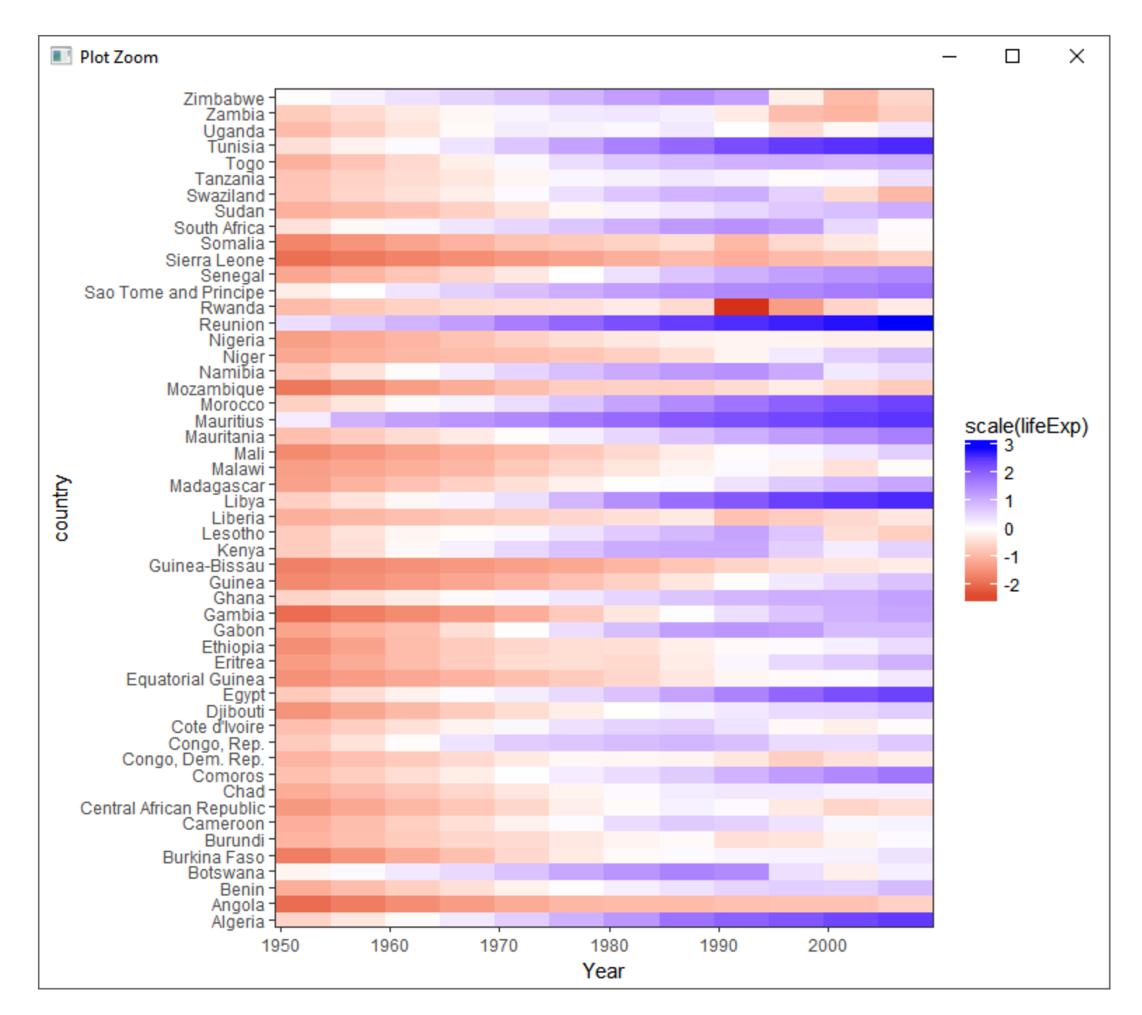


Figure 3.21 – HeatMap

3.10 Waterfall Chart

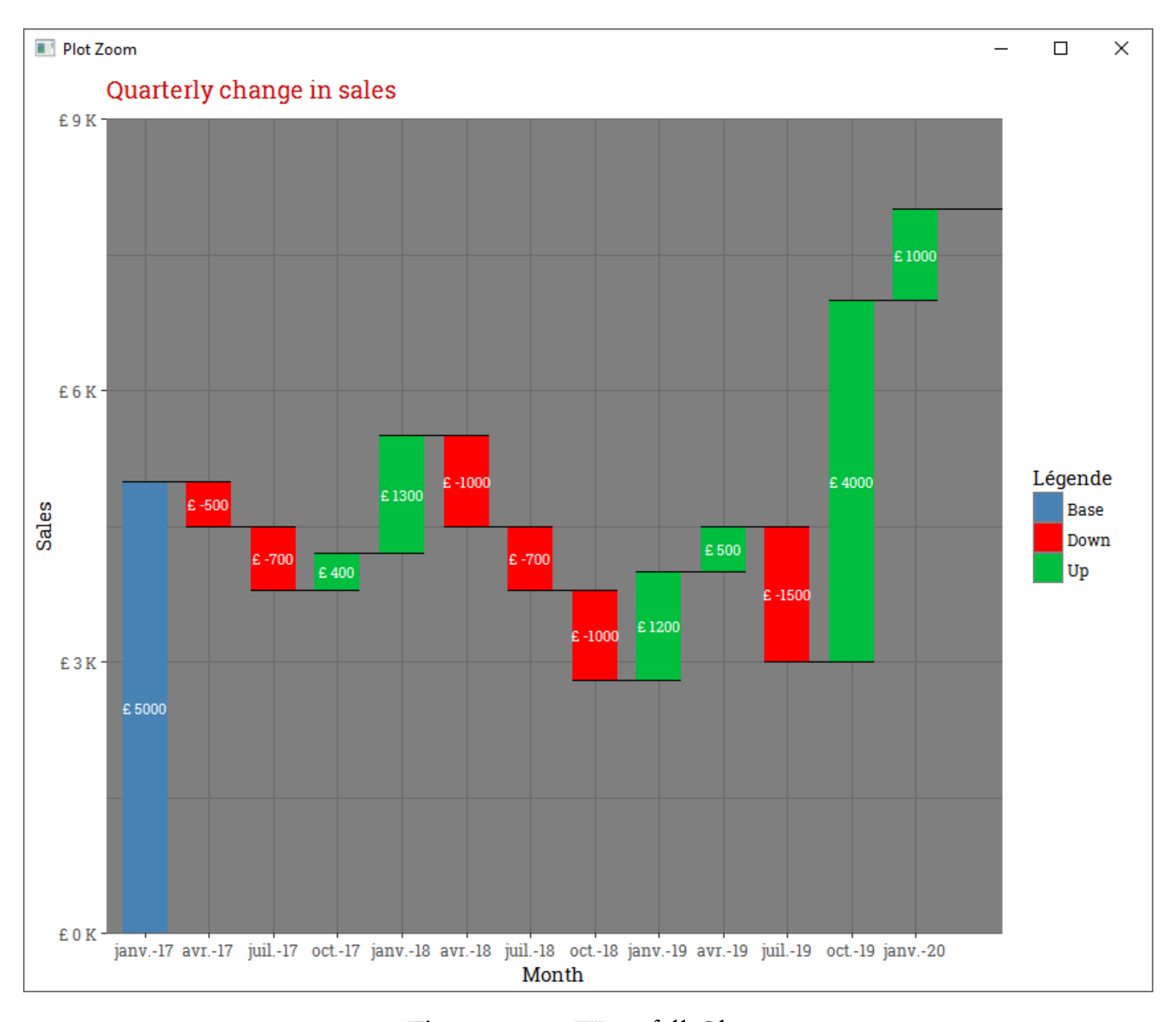


Figure 3.22 – Waterfall Chart

3.11 Diagramme en pyramide

Le diagramme en pyramide étant un cas particulier des diagrammes à barres la réalisation de la figure 3.23, repose essentiellement sur l'objet géométrique geom_bar() pour la réalisation des barres et geom_step() pour le tracé de ligne (cela ne concerne que la projection à l'horizon 2100).

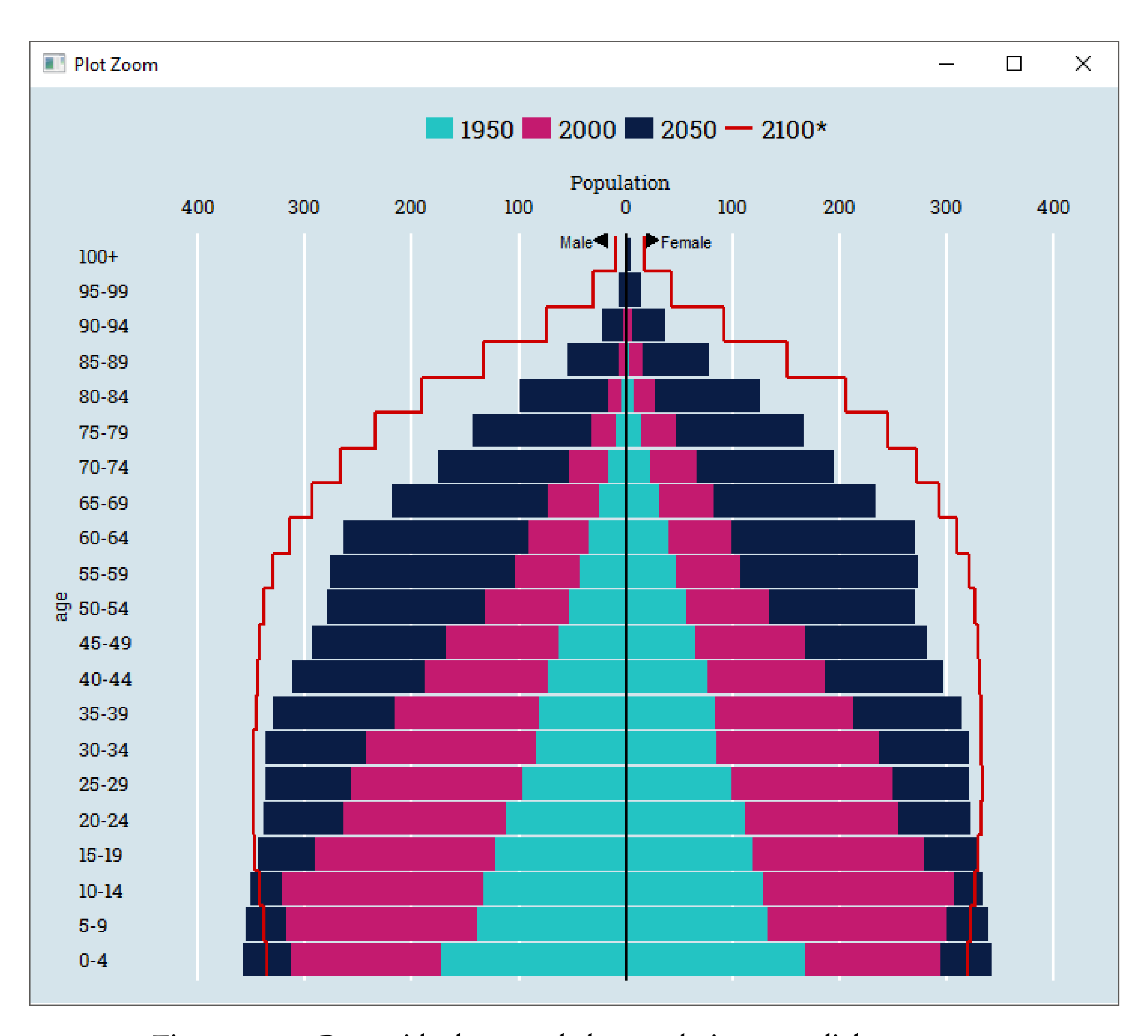


Figure 3.23 – Pyramide des ages de la population mondiale par genre

List of Figures

List of Tables

Index